# Contents

# Chapter 1

# Library B_Unification.intro

## 1.1 Introduction

## 1.2 Unification

Before defining what unification is, there is some terminology to understand. A *term* is either a variable or a function applied to terms [1]. By this definition, a constant term is just a nullary function. A *variable* is a symbol capable of taking on the value of any term. An example of a term is $f(a,\ x)$, where $f$ is a function of two arguments, $a$ is a constant, and $x$ is a variable. A term is *ground* if no variables occur in it [2]. The last example is not a ground term but $f(a,\ a)$ would be.

A *substitution* is a mapping from variables to terms. The *domain* of a substitution is the set of variables that do not get mapped to themselves. The *range* is the set of terms the are mapped to by the domain [2]. It is common for substitutions to be referred to as mappings from terms to terms. A substitution $\sigma$ can be extended to this form by defining $\hat{\sigma}(s)$ for two cases of $s$. If $s$ is a variable, then $\hat{\sigma}(s) := \sigma(s)$. If $s$ is a function $f(s_1, ..., s_n)$, then $\hat{\sigma}(s) := f(\hat{\sigma}(s_1), ..., \hat{\sigma}(s_n))$ [3].

Unification is the process of solving a set of equations between two terms. The set of equations is referred to as a *unification problem* [4]. The process of solving one of these problems can be classified by the set of terms considered and the equality of any two terms. The latter property is what distinguishes two broad groups of algorithms, namely syntactic and semantic unification. If two terms are only considered equal if they are identical, then the unification is *syntactic* [4]. If two terms are equal with respect to an equational theory, then the unification is *semantic* [5].

The goal of unification is to solve a problem, which means to produce a substitution that unifies all equations of a problem. A substitution $\sigma$ *unifies* an equation $s \stackrel{?}{=} t$ if applying $\sigma$ to both sides makes them equal $\sigma(s) = \sigma(t)$. If $\sigma$ unifies every equation in the problem S, we call $\sigma$ a *solution* or *unifier* of S [4].

The goal of a unification algorithm is not just to produce a unifier but to produce one that is most general. A substitution is a *most general unifier* or *mgu* of a problem if it is more general than every other solution to the problem. A substitution $\sigma$ is *more general* than $\sigma'$ if there exists a third substitution $\delta$ such that $\sigma'(u) = \delta(\sigma(u))$ for any term $u$ [4].

### 1.2.1 Syntatic Unification

This is the simpler version of unification. For two terms to be considered equal they must be identical. For example, the terms $x * y$ and $y * x$ are not syntactically equal, but would be equal modulo commutativity of multiplication. Problems of this kind can be solved by repeated transformations until the solution pops out similar to solving a linear system by Guassian elimination [6]. This version of unification is considered a simpler version of semantic unification because it is the special case where the set of equational identities is empty.

### 1.2.2 Semantic Unification

This kind of unification involves an equational theory. Given a set of identities $E$, we write that two terms $s$ and $t$ are equal with regards to $E$ as $s \approx_E t$. This means that identities of $E$ can be applied to $s$ as *s'* and $t$ as *t'* in some way to make them syntactically equal, *s'* = *t'*. As an example, let $C$ be the set $\{f(x, y) \approx f(y, x)\}$. This theory axiomatizes the commutativity of the function *f*. Knowing this, the problem $\{f(x, a) \overset{?}{=} f(a, b)\}$ is unified by the substitution $\{x \mapsto b\}$ since $f(b, a) \approx_C f(a, b)$. In general, for an arbitrary $E$, the problem of $E$-unification is undecidable [4].

### 1.2.3 Boolean Unification

In this paper, we focus on unfication modulo Boolean ring theory, also referred to as $B$-unification. The allowed terms in this theory are the constants 0 and 1 and binary functions $+$ and $*$. The set of identities $B$ is defined as the set $\{x + y \approx y + x, (x + y) + z \approx x + (y + z), x + x \approx 0, 0 + x \approx x, x * (y + z) \approx (x * y) + (x * z), x * y \approx y * x, (x * y) * z \approx xast(yastz), x * x \approx x, 0 * x \approx 0, 1 * x \approx x\}$ [7]. This set is equivalent to the theory of real numbers with the addition of $x + x \approx_B 0$ and $x * x \approx_B x$.

Although a unification problem is a set of equations between two terms, we will now show informally that a $B$-unification problem can be viewed as a single equation $t \overset{?}{\approx}_B 0$. Given a problem in its normal form $\{s_1 \overset{?}{\approx}_B t_1, ..., s_n \overset{?}{\approx}_B t_n\}$, we can transform it into $\{s_1 + t_1 \overset{?}{\approx}_B 0, ..., s_n + t_n \overset{?}{\approx}_B 0\}$ using a simple fact. The equation $s \approx_B t$ is equivalent to $s + t \approx_B 0$ since adding $t$ to both sides of the equation turns the right hand side into $\mathtt{t} + \mathtt{t}$ which simplifies to 0. Then, given a problem $\{t_1 \overset{?}{\approx}_B 0, ..., t_n \overset{?}{\approx}_B 0\}$, we can transform it into $\{(t_1 + 1) * ... * (t_n + 1) \overset{?}{\approx}_B 1\}$. Unifying both of these sets is equivalent because if any $t_1, ..., t_n$ is 1 the problem is not unifiable. Otherwise, if every $t_1, ..., t_n$ can be made to equal 0, then both problems will be solved.

## 1.3 Formal Verification

Formal verification is the term used to describe the act of verifying (or disproving) the correctness of software and hardware systems or theories. Formal verification consists of a set of techinques that perform static analysis on the behavior of a system, or the correctness of a theory. It differs to dynamic analysis that uses simulation to evaluate the correctness of a system.

Formal verification is used because it does not have to evaluate every possible case or state to determine if a system or theory meets all the preset logical conditions and rerquirements. Moreover, as design and software systems sizes have increased (along with their simulation times), verification teams have been looking for alternative methods of proving or disproving the correctness of a system in order to reduce the required time to perform a correctness check or evaluation.

### 1.3.1 Proof Assistance

A proof assistant is a software tool that is used to formulate and prove or disprove theorems in computer science or mathematical logic. They are also be called interactive theorem provers and they may also involve some type of proof and text editor that the user can use to form and prove and define theorems, lemmas , functions , etc. They facilitate that process by allowing the user to search definitions, terms and even provide some kind of guidance during the formulation or proof of a theorem.

### 1.3.2 Verifying Systems

### 1.3.3 Verifying Theories

## 1.4 Importance

## 1.5 Development

There are many different approaches that one could take to go about formalizing a proof of Boolean Unification algorithms, each with their own challenges. For this development, we have opted to base our work largely off chapter 10, *Equational Unification*, in *Term Rewriting and All That* by Franz Baader and Tobias Nipkow. Specifically, section 10.4, titled *Boolean Unification*, details Boolean rings, data structures to represent them, and two algorithms to perform unification in Boolean rings.

We chose to implement two data structures for representing the terms of a Boolean unification problem, and two algorithms for performing unification. The two data structures chosen are an inductive Term type and lists of lists representing polynomial-form terms. The two algorithms are Lowenheim\u2019s formula and successive variable elimination.

### 1.5.1 Data Structures

The data structure used to represent a Boolean unification problem completely changes the shape of both the unification algorithm and the proof of correctness, and is therefore a very important decision. For this development, we have selected two different representations of Boolean rings \u2013 first as a \u201cTerm\u201d inductive type, and then as lists of lists representing terms in polynomial form.

The Term inductive type, used in the proof of Lowenheim\u2019s algorithm, is very simple and rather intuitive \u2013 a term in a Boolean ring is one of 5 things:

- The number 0
- The number 1
- A variable
- Two terms added together
- Two terms multiplied together

In our development, variables are represented as natural numbers.

After defining terms like this, it is necessary to define a new equality relation, referred to as term equivalence, for comparing terms. With the term equivalence relation defined, it is easy to define ten axioms enabling the ten identities that hold true over terms in Boolean rings.

The inductive representation of terms in a Boolean ring is defined in the file *terms.v*. Unification over these terms is defined in *term_unif.v*.

The second representation, used in the proof of successive variable elimination, uses lists of lists of variables to represent terms in polynomial form. A monomial is a list of distinct variables multiplied together. A polynomial, then, is a list of distinct monomials added together. Variables are represented the same way, as natural numbers. The terms 0 and 1 are represented as the empty polynomial and the polynomial containing only the empty monomial, respectively.

The interesting part of the polynomial representation is how the ten identities are implemented. Rather than writing axioms enabling these transformations, we chose to implement the addition and multiplication operations in such a way to ensure these rules hold true, as described in *Term Rewriting*.

Addition is performed by cancelling out all repeated occurrences of monomials in the result of appending the two lists together (ie, x+x=0). This is equivalent to the symmetric difference in set theory, keeping only the terms that are in either one list or the other (but not both). Multiplication is slightly more complicated. The product of two polynomials is the result of multiplying all combinations of monomials in the two polynomials and removing all repeated monomials. The product of two monomials is the result of keeping only one copy of each repeated variable after appending the two together.

By defining the functions like this, and maintaining that the lists are sorted with no duplicates, we ensure that all 10 rules hold over the standard coq equivalence function. This of course has its own benefits and drawbacks, but lent itself better to the nature of successive variable elimination.

The polynomial representation is defined in the file *poly.v*. Unification over these polynomials is defined in *poly_unif.v*.

### 1.5.2 Algorithms

For unification algorithms, we once again followed the work laid out in *Term Rewriting and All That* and implemented both Lowenheim\u2019s algorithm and successive variable elimination.

The first solution, Lowenheim\u2019s algorithm, is built on top of the term inductive type. Lowenheim\u2019s is based on the idea that the Lowenheim formula can take a ground unifier of a Boolean unification problem and turn it into a most general unifier. The algorithm then of course first requires finding a ground solution, accomplished through brute force, which is then passed through the formula to create a most general unifier. Lowenheim\u2019s algorithm is implemented in the file *lowenheim.v*, and the proof of correctness is in *lowenheim_proof.v*.

The second algorithm, successive variable elimination, is built on top of the list-of-list polynomial approach. Successive variable elimination is built on the idea that by factoring variables out of the equation one-by-one, we can eventually reach a ground unifier. This unifier can then be built up with the variables that were previously eliminated until a most general unifier for the original unification problem is achieved. Successive variable elimination and its proof of correctness are both in *sve.v*.

# Chapter 2

# Library B_Unification.terms

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

## 2.1 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.

## 2.2 Terms

### 2.2.1 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 → term
| SUM : term → term → term
| PRODUCT : term → term → 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).
```

### 2.2.2 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.

Axiom T1_not_equiv_T0 :
  ~(T1 == T0).

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.
```

### 2.2.3 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 :
```

```
 ∀ x y z, x + (y + z) == (x + y) + z.
Proof.
  intros. rewrite sum_assoc. reflexivity.
Qed.

Lemma mul_assoc_opp :
 ∀ x y z, x × (y × z) == (x × y) × z.
Proof.
  intros. rewrite mul_assoc. reflexivity.
Qed.

Lemma distr_opp :
 ∀ x y z, x × y + x × z == x × ( y + z).
Proof.
  intros. rewrite distr. reflexivity.
Qed.
```

## 2.3 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.

### 2.3.1 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): var_set :=
  match vars with
    | nil ⇒ nil
    | n :: n' ⇒
    if (var_set_includes_var n n') then var_set_create_unique n'
    else n :: var_set_create_unique n'
  end.

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

Lemma vs_includes_true : ∀ (x : var) (lvar : list var),
  var_set_includes_var x lvar = true → In x lvar.
 Proof.
 intros.
  induction lvar.
  - simpl; intros.
  discriminate.
  - simpl in H. remember (beq_nat x a) as H2. destruct H2.
  + simpl. left. symmetry in HeqH2. pose proof beq_nat_true as H7. specialize (H7
x a HeqH2).
    symmetry in H7. apply H7.
  + specialize (IHlvar H). simpl. right. apply IHlvar.
Qed.

Lemma vs_includes_false : ∀ (x : var) (lvar : list var),
```

```
  var_set_includes_var x lvar = false → ¬ In x lvar.
 Proof.
 intros.
  induction lvar.
  - simpl; intros. unfold not. intros. destruct H0.
  - simpl in H. remember (beq_nat x a) as H2. destruct H2. inversion H.
    specialize (IHlvar H). firstorder. intuition. apply IHlvar. simpl in H0.
    destruct H0.
    { inversion HeqH2. symmetry in H2. pose proof beq_nat_false as H7. specialize
(H7 x a H2).
      rewrite H0 in H7. destruct H7. intuition. }
    { apply H0. }
Qed.

Lemma in_dup_and_non_dup :
 ∀ (x: var) (lvar : list var),
 In x lvar ↔ In x (var_set_create_unique lvar).
Proof.
 intros. split.
 - induction lvar.
  + intros. simpl in H. destruct H.
  + intros. simpl. remember(var_set_includes_var a lvar) as C. destruct C.
   { symmetry in HeqC. pose proof vs_includes_true as H7. specialize (H7 a lvar HeqC).
     simpl in H. destruct H.
    { rewrite H in H7. specialize (IHlvar H7). apply IHlvar. }
    { specialize (IHlvar H). apply IHlvar. }
   }
   { symmetry in HeqC. pose proof vs_includes_false as H7. specialize (H7 a lvar HeqC).
     simpl in H. destruct H.
    { simpl. left. apply H. }
    { specialize (IHlvar H). simpl. right. apply IHlvar. }
   }
 - induction lvar.
   + intros. simpl in H. destruct H.
   + intros. simpl in H. remember(var_set_includes_var a lvar) as C. destruct C.
     { symmetry in HeqC. pose proof vs_includes_true as H7. specialize (H7 a lvar
HeqC).
      specialize (IHlvar H). simpl. right. apply IHlvar. }
     { symmetry in HeqC. pose proof vs_includes_false as H7. specialize (H7 a lvar
HeqC).
       simpl in H. destruct H.
      { simpl. left. apply H. }
      { specialize (IHlvar H). simpl. right. apply IHlvar. } }
```

```
Qed.
```

### 2.3.2 Examples

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

## 2.4 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.

### 2.4.1 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.
```

### 2.4.2 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 :
  ∀ 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 :
  ∀ x, (ground_term x) → (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.
```

### 2.4.3 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 × T1)) → False.
Proof.
simpl. intros. destruct H. apply H.
Qed.
```

## 2.5 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.

### 2.5.1 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 :=
  ∀ t, apply_subst t s1 == apply_subst t s2.
```

```
Definition subst_is_id_subst (t : term) (s : subst) : Prop :=
  (apply_subst t s) == t.
```

### 2.5.2 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.

Given that we have a definition for identity substitutions, we should prove that identity substitutions do not modify a term.

```
Lemma id_subst:
  ∀ (t : term) (l : var_set),
  apply_subst t (build_id_subst l) == t.
Proof.
```

```
intros. induction t.
{
  simpl. reflexivity.
}
{
  simpl. reflexivity.
}
{
  simpl. induction l.
  {
    simpl. reflexivity.
  }
  {
    simpl. destruct (beq_nat a v) eqn: e.
    {
      apply beq_nat_true in e. rewrite e. reflexivity.
    }
    {
      apply IHl.
    }
  }
}
{
  simpl. rewrite IHt1. rewrite IHt2. reflexivity.
}
{
  simpl. rewrite IHt1. rewrite IHt2. reflexivity.
}
Qed.
```

```
Lemma apply_subst_compat : ∀ (t t' : term),
     t == t' → ∀ (sigma: subst), (apply_subst t sigma) == (apply_subst t' sigma).
Proof.
intros. induction t.
  - induction t'.
    + simpl. reflexivity.
    + simpl. apply H.
    + simpl. rewrite H.
Admitted.
```

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

Qed.

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 :
  $\forall$ $x$, (ground_term $x$) $\to$ ($\forall$ $s$, 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.

A fundamental property of substitutions is their distributivity and associativity across the summation and multiplication of terms. Again the importance of these proofs will not become apparent until we talk about unification.

Lemma subst_distribution :
  $\forall$ $s$ $x$ $y$, 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 :
  $\forall$ $s$ $x$ $y$, apply_subst $x$ $s$ $\times$ apply_subst $y$ $s$ == apply_subst $(x \times y)$ $s$.
Proof.
intro. induction $s$. intros. reflexivity. intros. simpl. reflexivity.
Qed.

Lemma subst_sum_distr_opp :
  $\forall$ $s$ $x$ $y$, 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 :
  $\forall$ $s$ $x$ $y$, apply_subst $(x \times y)$ $s$ == apply_subst $x$ $s$ $\times$ 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.
intros. simpl. destruct (beq_nat v v) eqn: e. apply beq_nat_true in e.
reflexivity. apply beq_nat_false in e. firstorder.
Qed.
```

### 2.5.3 Examples

Here are some examples showcasing the nature of applying substitutions to terms.

```
Example subst_ex1 :
  (apply_subst (T0 + T1) []) == T0 + T1.
Proof.
intros. reflexivity.
Qed.

Example subst_ex2 :
  (apply_subst (VAR 0 × VAR 1) [(0, T0)]) == T0.
Proof.
intros. simpl. apply mul_T0_x.
Qed.
```

## 2.6 Unification

Now that we have established the concept of term substitutions in Coq, it is time for us to formally define the concept of Boolean unification. Unification, in its most literal sense, refers to the act of applying a substitution to terms in order to make them equivalent to each other. In other words, to say that two terms are unifiable is to really say that there exists a substitution such that the two terms are equal. Interestingly enough, we can abstract this concept further to simply saying that a single term is unifiable if there exists a substitution such that the term will be equivalent to 0. By doing this abstraction, we can prove that equation solving and unification are essentially the same fundamental problem.

Below is the initial definition for unification, namely that two terms can be unified to be equivalent to one another. By starting here we will show each step towards abstracting unification to refer to a single term.

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

Here is a simple example demonstrating the concept of testing whether two terms are unified by a substitution.

```
Example ex_unif1 :
  unifies (VAR 0) (VAR 1) ((0, T1) :: (1, T1) :: nil).
Proof.
unfold unifies. simpl. reflexivity.
Qed.
```

Now we are going to show that moving both terms to one side of the equivalence relation through addition does not change the concept of unification.

```
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.
```

Now we can define what it means for a substitution to be a unifier for a given term.

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

Example unifier_ex1 :
  (unifier (VAR 0) ((0, T0) :: nil)).
Proof.
unfold unifier. simpl. reflexivity.
Qed.
```

To ensure our efforts were not in vain, let us now prove that this last abstraction of the unification problem is still equivalent to the original.

```
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.
```

Lastly let us define a term to be unifiable if there exists a substitution that unifies it.

```
Definition unifiable (t : term) : Prop :=
  ∃ s, unifier t s.

Example unifiable_ex1 :
  ∃ x, unifiable (x + T1).
Proof.
∃ (T1). unfold unifiable. unfold unifier.
∃ nil. simpl. rewrite sum_x_x. reflexivity.
Qed.
```

## 2.7 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 ).

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 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 most_general_unifier_compat : ∀ (t t' : term),
      t == t' → ∀ (sigma: subst), (most_general_unifier t sigma) ↔ (most_general_unifier t'
sigma).
Proof.
Admitted.
```

## 2.8 Auxilliary Computational Operations and Simplifications

These functions below will come in handy later during the Lowenheim formula proof.

```
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, T0 ⇒ T0
    | T0, T1 ⇒ T1
    | T1, T0 ⇒ T1
    | T1 , T1 ⇒ T0
    | _ , _ ⇒ SUM a b
  end.
```

```
Definition mult_one_step (a b : term) : term :=
```

```
  match a, b with
    | T0, T0 ⇒ T0
    | T0, T1 ⇒ T0
    | T1, T0 ⇒ T0
    | T1 , T1 ⇒ T1
    | _ , _ ⇒ PRODUCT 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.
Lemma pos_left_sum_compat : ∀ (t t1 t2 : term),
      t == t1 → plus_one_step t1 t2 == plus_one_step t t2.
Proof.
  intros. induction t1.
  - induction t.
    + reflexivity.
    + apply T1_not_equiv_T0 in H. inversion H.
    + induction t2.
      { simpl. rewrite H. rewrite sum_x_x. reflexivity. }
      { simpl. rewrite H. rewrite sum_id. reflexivity. }
      { simpl. rewrite H. reflexivity. }
      { simpl. rewrite H. reflexivity. }
      { simpl. rewrite H. reflexivity. }
    + induction t2.
      { simpl. rewrite H. rewrite sum_x_x. reflexivity. }
      { simpl. rewrite H. rewrite sum_id. reflexivity. }
      { simpl. rewrite H. reflexivity. }
      { simpl. rewrite H. reflexivity. }
      { simpl. rewrite H. reflexivity. }
    + induction t2.
      { simpl. rewrite H. rewrite sum_x_x. reflexivity. }
      { simpl. rewrite H. rewrite sum_id. reflexivity. }
      { simpl. rewrite H. reflexivity. }
      { simpl. rewrite H. reflexivity. }
      { simpl. rewrite H. reflexivity. }
  - induction t.
    + induction t2.
```

```
    { simpl. rewrite H. reflexivity. }
    { simpl. rewrite H. reflexivity. }
    { simpl. rewrite H. reflexivity. }
    { simpl. rewrite H. reflexivity. }
    { simpl. rewrite H. reflexivity. }
  + induction t2.
    { simpl. reflexivity. }
    { simpl. reflexivity. }
    { simpl. reflexivity. }
    { simpl. reflexivity. }
    { simpl. reflexivity. }
  + induction t2.
    { simpl. rewrite H. rewrite sum_comm. rewrite sum_id. reflexivity. }
    { simpl. rewrite H. rewrite sum_x_x. reflexivity. }
    { simpl. rewrite H. reflexivity. }
    { simpl. rewrite H. reflexivity. }
    { simpl. rewrite H. reflexivity. }
  + induction t2.
    { simpl. rewrite H. rewrite sum_comm. rewrite sum_id. reflexivity. }
    { simpl. rewrite H. rewrite sum_x_x. reflexivity. }
    { simpl. rewrite H. reflexivity. }
    { simpl. rewrite H. reflexivity. }
    { simpl. rewrite H. reflexivity. }
  + induction t2.
    { simpl. rewrite H. rewrite sum_comm. rewrite sum_id. reflexivity. }
    { simpl. rewrite H. rewrite sum_x_x. reflexivity. }
    { simpl. rewrite H. reflexivity. }
    { simpl. rewrite H. reflexivity. }
    { simpl. rewrite H. reflexivity. }
- induction t.
  + induction t2.
    { simpl. rewrite H. rewrite sum_x_x. rewrite H. reflexivity. }
    { simpl. rewrite <- H. rewrite sum_id. reflexivity. }
    { simpl. rewrite H. reflexivity. }
    { simpl. rewrite H. reflexivity. }
    { simpl. rewrite H. reflexivity. }
  + induction t2.
    { simpl. rewrite <- H. rewrite sum_comm. rewrite sum_id. reflexivity. }
    { simpl. rewrite H. rewrite sum_x_x. reflexivity. }
    { simpl. rewrite H. reflexivity. }
    { simpl. rewrite H. reflexivity. }
    { simpl. rewrite H. reflexivity. }
```

```
    + induction t2.
      { simpl. rewrite H. reflexivity. }
      { simpl. rewrite H. reflexivity. }
      { simpl. rewrite H. reflexivity. }
      { simpl. rewrite H. reflexivity. }
      { simpl. rewrite H. reflexivity. }
    + induction t2.
      { simpl. rewrite H. reflexivity. }
      { simpl. rewrite H. reflexivity. }
      { simpl. rewrite H. reflexivity. }
      { simpl. rewrite H. reflexivity. }
      { simpl. rewrite H. reflexivity. }
    + induction t2.
      { simpl. rewrite H. reflexivity. }
      { simpl. rewrite H. reflexivity. }
      { simpl. rewrite H. reflexivity. }
      { simpl. rewrite H. reflexivity. }
      { simpl. rewrite H. reflexivity. }
  - induction t.
    + induction t2.
      { simpl. rewrite ← H. rewrite sum_x_x. reflexivity. }
      { simpl. rewrite ← H. rewrite sum_id. reflexivity. }
      { simpl. rewrite ← H. reflexivity. }
      { simpl. rewrite ← H. reflexivity. }
      { simpl. rewrite ← H. reflexivity. }
    + induction t2.
      { simpl. rewrite ← H. rewrite sum_comm. rewrite sum_id. reflexivity. }
      { simpl. rewrite H. rewrite sum_x_x. reflexivity. }
      { simpl. rewrite ← H. reflexivity. }
      { simpl. rewrite ← H. reflexivity. }
      { simpl. rewrite ← H. reflexivity. }
    + induction t2.
      { simpl. rewrite H. reflexivity. }
      { simpl. rewrite H. reflexivity. }
      { simpl. rewrite H. reflexivity. }
      { simpl. rewrite H. reflexivity. }
      { simpl. rewrite H. reflexivity. }
    + induction t2.
      { simpl. rewrite H. reflexivity. }
      { simpl. rewrite H. reflexivity. }
      { simpl. rewrite H. reflexivity. }
      { simpl. rewrite H. reflexivity. }
```

```
      { simpl. rewrite H. reflexivity. }
    + induction t2.
      { simpl. rewrite H. reflexivity. }
      { simpl. rewrite H. reflexivity. }
      { simpl. rewrite H. reflexivity. }
      { simpl. rewrite H. reflexivity. }
      { simpl. rewrite H. reflexivity. }
  - induction t.
    + induction t2.
      { simpl. rewrite ← H. rewrite sum_x_x. reflexivity. }
      { simpl. rewrite ← H. rewrite sum_id. reflexivity. }
      { simpl. rewrite ← H. reflexivity. }
      { simpl. rewrite ← H. reflexivity. }
      { simpl. rewrite ← H. reflexivity. }
    + induction t2.
      { simpl. rewrite ← H. rewrite sum_comm. rewrite sum_id. reflexivity. }
      { simpl. rewrite H. rewrite sum_x_x. reflexivity. }
      { simpl. rewrite ← H. reflexivity. }
      { simpl. rewrite ← H. reflexivity. }
      { simpl. rewrite ← H. reflexivity. }
    + induction t2.
      { simpl. rewrite H. reflexivity. }
      { simpl. rewrite H. reflexivity. }
      { simpl. rewrite H. reflexivity. }
      { simpl. rewrite H. reflexivity. }
      { simpl. rewrite H. reflexivity. }
    + induction t2.
      { simpl. rewrite H. reflexivity. }
      { simpl. rewrite H. reflexivity. }
      { simpl. rewrite H. reflexivity. }
      { simpl. rewrite H. reflexivity. }
      { simpl. rewrite H. reflexivity. }
    + induction t2.
      { simpl. rewrite H. reflexivity. }
      { simpl. rewrite H. reflexivity. }
      { simpl. rewrite H. reflexivity. }
      { simpl. rewrite H. reflexivity. }
      { simpl. rewrite H. reflexivity. }
Qed.

Lemma pos_right_sum_compat : ∀ (t t1 t2 : term),
    t == t2 → plus_one_step t1 t2 == plus_one_step t1 t.
Proof.
```

```
intros. induction t1.
  - induction t.
    + induction t2.
      { simpl. reflexivity. }
      { simpl. rewrite H. reflexivity. }
      { simpl. rewrite H. rewrite sum_x_x. apply H. }
      { simpl. rewrite <- H. rewrite sum_x_x. reflexivity. }
      { simpl. rewrite <- H. rewrite sum_x_x. reflexivity. }
    + induction t2.
      { simpl. rewrite H. reflexivity. }
      { simpl. reflexivity. }
      { simpl. rewrite H. rewrite sum_id. reflexivity. }
      { simpl. rewrite H. rewrite sum_id. reflexivity. }
      { simpl. rewrite <- H. rewrite sum_id. reflexivity. }
    + induction t2.
      { simpl. rewrite H. rewrite sum_x_x. reflexivity. }
      { simpl. rewrite H. rewrite sum_id. reflexivity. }
      { simpl. rewrite H. rewrite sum_id. reflexivity. }
      { simpl. rewrite H. rewrite sum_id. reflexivity. }
      { simpl. rewrite <- H. rewrite sum_id. reflexivity. }
    + induction t2.
      { simpl. rewrite H. rewrite sum_x_x. reflexivity. }
      { simpl. rewrite H. rewrite sum_id. reflexivity. }
      { simpl. rewrite H. rewrite sum_id. reflexivity. }
      { simpl. rewrite H. rewrite sum_id. reflexivity. }
      { simpl. rewrite <- H. rewrite sum_id. reflexivity. }
    + induction t2.
      { simpl. rewrite H. rewrite sum_x_x. reflexivity. }
      { simpl. rewrite H. rewrite sum_id. reflexivity. }
      { simpl. rewrite H. rewrite sum_id. reflexivity. }
      { simpl. rewrite H. rewrite sum_id. reflexivity. }
      { simpl. rewrite <- H. rewrite sum_id. reflexivity. }
  - induction t.
    + induction t2.
      { simpl. reflexivity. }
      { simpl. rewrite H. reflexivity. }
      { simpl. rewrite <- H. rewrite sum_comm. rewrite sum_id. reflexivity. }
      { simpl. rewrite <- H. rewrite sum_comm. rewrite sum_id. reflexivity. }
      { simpl. rewrite <- H. rewrite sum_comm. rewrite sum_id. reflexivity. }
    + induction t2.
      { simpl. rewrite H. reflexivity. }
      { simpl. reflexivity. }
```

```
    { simpl. rewrite H. rewrite sum_x_x. reflexivity. }
    { simpl. rewrite H. rewrite sum_x_x. reflexivity. }
    { simpl. rewrite ← H. rewrite sum_x_x. reflexivity. }
  + induction t2.
    { simpl. rewrite H. rewrite sum_comm. rewrite sum_id. reflexivity. }
    { simpl. rewrite H. rewrite sum_x_x. reflexivity. }
    { simpl. rewrite H. reflexivity. }
    { simpl. rewrite H. reflexivity. }
    { simpl. rewrite ← H. reflexivity. }
  + induction t2.
    { simpl. rewrite H. rewrite sum_comm. rewrite sum_id. reflexivity. }
    { simpl. rewrite H. rewrite sum_x_x. reflexivity. }
    { simpl. rewrite H. reflexivity. }
    { simpl. rewrite H. reflexivity. }
    { simpl. rewrite ← H. reflexivity. }
  + induction t2.
    { simpl. rewrite H. rewrite sum_comm. rewrite sum_id. reflexivity. }
    { simpl. rewrite H. rewrite sum_x_x. reflexivity. }
    { simpl. rewrite H. reflexivity. }
    { simpl. rewrite H. reflexivity. }
    { simpl. rewrite ← H. reflexivity. }
- induction t.
  + induction t2.
    { simpl. reflexivity. }
    { simpl. rewrite H. reflexivity. }
    { simpl. rewrite ← H. reflexivity. }
    { simpl. rewrite ← H. reflexivity. }
    { simpl. rewrite ← H. reflexivity. }
  + induction t2.
    { simpl. rewrite H. reflexivity. }
    { simpl. reflexivity. }
    { simpl. rewrite H. reflexivity. }
    { simpl. rewrite H. reflexivity. }
    { simpl. rewrite ← H. reflexivity. }
  + induction t2.
    { simpl. rewrite H. reflexivity. }
    { simpl. rewrite H. reflexivity. }
    { simpl. rewrite H. reflexivity. }
    { simpl. rewrite H. reflexivity. }
    { simpl. rewrite ← H. reflexivity. }
  + induction t2.
    { simpl. rewrite H. reflexivity. }
```

```
    { simpl. rewrite H. reflexivity. }
    { simpl. rewrite H. reflexivity. }
    { simpl. rewrite H. reflexivity. }
    { simpl. rewrite <- H. reflexivity. }
  + induction t2.
    { simpl. rewrite H. reflexivity. }
    { simpl. rewrite H. reflexivity. }
    { simpl. rewrite H. reflexivity. }
    { simpl. rewrite H. reflexivity. }
    { simpl. rewrite <- H. reflexivity. }
- induction t.
  + induction t2.
    { simpl. reflexivity. }
    { simpl. rewrite H. reflexivity. }
    { simpl. rewrite <- H. reflexivity. }
    { simpl. rewrite <- H. reflexivity. }
    { simpl. rewrite <- H. reflexivity. }
  + induction t2.
    { simpl. rewrite H. reflexivity. }
    { simpl. reflexivity. }
    { simpl. rewrite H. reflexivity. }
    { simpl. rewrite H. reflexivity. }
    { simpl. rewrite <- H. reflexivity. }
  + induction t2.
    { simpl. rewrite H. reflexivity. }
    { simpl. rewrite H. reflexivity. }
    { simpl. rewrite H. reflexivity. }
    { simpl. rewrite H. reflexivity. }
    { simpl. rewrite <- H. reflexivity. }
  + induction t2.
    { simpl. rewrite H. reflexivity. }
    { simpl. rewrite H. reflexivity. }
    { simpl. rewrite H. reflexivity. }
    { simpl. rewrite H. reflexivity. }
    { simpl. rewrite <- H. reflexivity. }
  + induction t2.
    { simpl. rewrite H. reflexivity. }
    { simpl. rewrite H. reflexivity. }
    { simpl. rewrite H. reflexivity. }
    { simpl. rewrite H. reflexivity. }
    { simpl. rewrite <- H. reflexivity. }
- induction t.
```

```
  + induction t2.
    { simpl. reflexivity. }
    { simpl. rewrite H. reflexivity. }
    { simpl. rewrite ← H. reflexivity. }
    { simpl. rewrite ← H. reflexivity. }
    { simpl. rewrite ← H. reflexivity. }
  + induction t2.
    { simpl. rewrite H. reflexivity. }
    { simpl. reflexivity. }
    { simpl. rewrite H. reflexivity. }
    { simpl. rewrite H. reflexivity. }
    { simpl. rewrite ← H. reflexivity. }
  + induction t2.
    { simpl. rewrite H. reflexivity. }
    { simpl. rewrite H. reflexivity. }
    { simpl. rewrite H. reflexivity. }
    { simpl. rewrite H. reflexivity. }
    { simpl. rewrite ← H. reflexivity. }
  + induction t2.
    { simpl. rewrite H. reflexivity. }
    { simpl. rewrite H. reflexivity. }
    { simpl. rewrite H. reflexivity. }
    { simpl. rewrite H. reflexivity. }
    { simpl. rewrite ← H. reflexivity. }
  + induction t2.
    { simpl. rewrite H. reflexivity. }
    { simpl. rewrite H. reflexivity. }
    { simpl. rewrite H. reflexivity. }
    { simpl. rewrite H. reflexivity. }
    { simpl. rewrite ← H. reflexivity. }
Qed.

Lemma pos_left_mul_compat : ∀ (t t1 t2 : term),
    t == t1 → mult_one_step t1 t2 == mult_one_step t t2.
Proof.
Admitted.

Lemma pos_right_mul_compat : ∀ (t t1 t2 : term),
    t == t2 → mult_one_step t1 t2 == mult_one_step t1 t.
Proof.
Admitted.

Lemma simplify_eqv :
 ∀ (t : term),
 simplify t == t.
```

```
Proof.
 intros. induction t.
 - simpl. reflexivity.
 - simpl. reflexivity.
 - simpl. reflexivity.
 - simpl. pose proof pos_left_sum_compat. specialize (H t1 (simplify t1) (simplify t2)).
    symmetry in IHt1. specialize (H IHt1). rewrite H.
   pose proof pos_right_sum_compat. specialize (H0 (simplify t2) t1 t2).
   specialize (H0 IHt2). symmetry in H0. rewrite H0.
   induction t1.
   + induction t2.
     { simpl. rewrite sum_x_x. reflexivity. }
     { simpl. rewrite sum_id. reflexivity. }
     { simpl. reflexivity. }
     { simpl. reflexivity. }
     { simpl. reflexivity. }
   + induction t2.
     { simpl. rewrite sum_id_sym. reflexivity. }
     { simpl. rewrite sum_x_x. reflexivity. }
     { simpl. reflexivity. }
     { simpl. reflexivity. }
     { simpl. reflexivity. }
   + simpl. reflexivity.
   + simpl. reflexivity.
   + simpl. reflexivity.
 - simpl. pose proof pos_left_mul_compat. specialize (H t1 (simplify t1) (simplify t2)).
    symmetry in IHt1. specialize (H IHt1). rewrite H.
   pose proof pos_right_mul_compat. specialize (H0 (simplify t2) t1 t2).
   specialize (H0 IHt2). symmetry in H0. rewrite H0.
   induction t1.
   + induction t2.
     { simpl. rewrite mul_x_x. reflexivity. }
     { simpl. rewrite mul_T0_x. reflexivity. }
     { simpl. reflexivity. }
     { simpl. reflexivity. }
     { simpl. reflexivity. }
   + induction t2.
     { simpl. rewrite mul_T0_x_sym. reflexivity. }
     { simpl. rewrite mul_x_x. reflexivity. }
     { simpl. reflexivity. }
     { simpl. reflexivity. }
     { simpl. reflexivity. }
```

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

# Chapter 3

# 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 4

# Library B_Unification.lowenheim_proof

```
Require Export lowenheim_formula.
Require Export EqNat.
Require Import List.
Import ListNotations.
Import Coq.Init.Tactics.
Require Export Classical_Prop.
```

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.
 intros. firstorder.
Qed.

Lemma term_vars_distr :
∀ (t1 t2 : term),
 (term_vars (t1 + t2)) = (term_vars t1 ) ++ (term_vars t2).
Proof.
 intros.
 induction t2.
 - simpl. reflexivity.
 - simpl. reflexivity.
 - simpl. reflexivity.
 - simpl. reflexivity.
```

```
 - simpl. reflexivity.
Qed.

Lemma tv_h1:
∀ (t1 t2 : term) ,
∀ (x : var),
 (In x (term_vars t1)) → (In x (term_vars (t1 + t2))).
Proof.
intros. induction t2.
 - simpl. rewrite app_nil_r. apply H.
 - simpl. rewrite app_nil_r. apply H.
 - simpl. pose proof in_or_app as H1. specialize (H1 var (term_vars t1) [v] x).
firstorder.
 - rewrite term_vars_distr. apply in_or_app. left. apply H.
 - rewrite term_vars_distr. apply in_or_app. left. apply H.
Qed.

Lemma tv_h2:
∀ (t1 t2 : term) ,
∀ (x : var),
 (In x (term_vars t2)) → (In x (term_vars (t1 + t2))).
Proof.
intros. induction t1.
 - simpl. apply H.
 - simpl. apply H.
 - simpl. pose proof in_or_app as H1. right. apply H.
 - rewrite term_vars_distr. apply in_or_app. right. apply H.
 - rewrite term_vars_distr. apply in_or_app. right. apply H.
Qed.

Lemma helper_2a:
  ∀ (t1 t2 t' : term),
  sub_term (t1 + t2) t' → sub_term t1 t'.
Proof.
 intros. unfold sub_term in *. intros. specialize (H x).
 pose proof in_dup_and_non_dup as H10. unfold term_unique_vars. unfold term_unique_vars
in *.
 pose proof tv_h1 as H7. apply H. specialize (H7 t1 t2 x). specialize (H10 x
(term_vars (t1 + t2))). destruct H10 .
 apply H1. apply H7. pose proof in_dup_and_non_dup as H10. specialize (H10 x
(term_vars t1)). destruct H10.
 apply H4. apply H0.
Qed.

Lemma helper_2b:
```

```
  ∀ (t1 t2 t' : term),
  sub_term (t1 + t2) t' → sub_term t2 t'.
Proof.
intros. unfold sub_term in *. intros. specialize (H x).
pose proof in_dup_and_non_dup as H10. unfold term_unique_vars. unfold term_unique_vars
in *.
 pose proof tv_h2 as H7.  apply H. specialize (H7 t1 t2 x).  specialize (H10 x
(term_vars (t1 + t2))). destruct H10 .
 apply H1.  apply H7.  pose proof in_dup_and_non_dup as H10.  specialize (H10 x
(term_vars t2)). destruct H10.
 apply H4. apply H0.
Qed.

Lemma elt_in_list:
 ∀ (x: var) (a : var) (l : list var),
  (In x (a::l)) →
  x = a ∨ (In x l).
Proof.
intros.
pose proof in_inv as H1.
specialize (H1 var a x l H).
destruct H1.
 - left. symmetry in H0. apply H0.
 - right. apply H0.
Qed.

Lemma elt_not_in_list:
 ∀ (x: var) (a : var) (l : list var),
  ¬ (In x (a::l)) →
  x ≠ a ∧ ¬ (In x l).
Proof.
intros.
pose proof not_in_cons. specialize (H0 var x a l). destruct H0.
specialize (H0 H). apply H0.
Qed.

Lemma in_list_of_var_term_of_var:
∀ (x : var),
  In x (term_unique_vars (VAR x)).
Proof.
intros. simpl. left. intuition.
Qed.

Lemma var_in_out_list:
  ∀ (x : var) (lvar : list var),
```

```
  (In x lvar) ∨ ¬ (In x lvar).
Proof.
 intros.
 pose proof classic as H1. specialize (H1 (In x lvar)). apply H1.
Qed.
```

3.2 Proof that Lownheim's algorithm unifes a given term

```
Lemma helper1_easy:
 ∀ (x: var) (lvar : list var) (sig1 sig2 : subst) (s : term),
 (In x lvar) →
  apply_subst (VAR x) (build_on_list_of_vars lvar s sig1 sig2)
  ==
  apply_subst (VAR x) (build_on_list_of_vars (cons x nil) s sig1 sig2).
Proof.
 intros.
 induction lvar.
 - simpl. simpl in H. destruct H.
 - apply elt_in_list in H. destruct H.
  + simpl. destruct (beq_nat a x) as []eqn:?.
   { apply beq_nat_true in Heqb. destruct (beq_nat x x) as []eqn:?.
    { rewrite H. reflexivity. }
    { apply beq_nat_false in Heqb.
      { destruct Heqb. }
      { rewrite Heqb. apply Heqb0. } }}
   { simpl in IHlvar. apply IHlvar. symmetry in H. rewrite H in Heqb.
    apply beq_nat_false in Heqb. destruct Heqb. intuition. }
  + destruct (beq_nat a x) as []eqn:?.
    { apply beq_nat_true in Heqb. symmetry in Heqb. rewrite Heqb in IHlvar. rewrite
Heqb.
        simpl in IHlvar. simpl. destruct (beq_nat a a) as []eqn:?.
     { reflexivity. }
     { apply IHlvar. rewrite Heqb in H. apply H. }}
    { apply beq_nat_false in Heqb. simpl. destruct (beq_nat a x) as []eqn:?.
     { apply beq_nat_true in Heqb0. rewrite Heqb0 in Heqb. destruct Heqb. intuition.
}
     { simpl in IHlvar. apply IHlvar. apply H. }}
Qed.

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.
 intros. unfold sub_term in H. specialize (H v). pose proof in_list_of_var_term_of_var
as H3.
 specialize (H3 v). specialize (H H3). pose proof helper1_easy as H2.
 specialize (H2 v (term_unique_vars t') sig1 sig2 s). apply H2. apply H.
Qed.

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

```
Lemma lowenheim_rephrase1_easy :
  ∀ (l : list var) (x : var) (sig1 : subst) (sig2 : subst) (s : term),
  (In x l) →
  (apply_subst (VAR x) (build_on_list_of_vars l s sig1 sig2)) ==
  (s + T1) × (apply_subst (VAR x) sig1 ) + s × (apply_subst (VAR x) sig2 ).
Proof.
intros.
induction l.
- simpl. unfold In in H. destruct H.
- apply elt_in_list in H. destruct H.
  + simpl. destruct (beq_nat a x) as []eqn:?.
    { rewrite H. reflexivity. }
    { pose proof beq_nat_false as H2. specialize (H2 a x).
      specialize (H2 Heqb). intuition. symmetry in H. specialize (H2 H). inversion
H2. }
  + simpl. destruct (beq_nat a x) as []eqn:?.
    { symmetry in Heqb. pose proof beq_nat_eq as H2. specialize (H2 a x). specialize
(H2 Heqb). rewrite H2.
      reflexivity. }
    { apply IHl. apply H. }
Qed.

Lemma helper_3a:
∀ (x: var) (l: list var),
In x l →
  apply_subst (VAR x) (build_id_subst l) == VAR x.
Proof.
intros. induction l.
 - unfold build_id_subst. simpl. reflexivity.
 - apply elt_in_list in H. destruct H.
   + simpl. destruct (beq_nat a x) as []eqn:?.
    { rewrite H. reflexivity. }
```

```
    { pose proof beq_nat_false as H2. specialize (H2 a x).
       specialize (H2 Heqb). intuition. symmetry in H. specialize (H2 H). inversion
H2. }
   + simpl. destruct (beq_nat a x) as []eqn:?.
    { symmetry in Heqb. pose proof beq_nat_eq as H2. specialize (H2 a x). specialize
(H2 Heqb). rewrite H2.
      reflexivity. }
    { apply IHl. apply H. }
Qed.

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.
  unfold build_lowenheim_subst. pose proof lowenheim_rephrase1_easy as H1.
  specialize (H1 (term_unique_vars t) x (build_id_subst (term_unique_vars t)) tau t).
  rewrite helper_3a in H1.
 - apply H1. apply H0.
 - apply H0.
Qed.

Lemma lowenheim_rephrase2_easy :
  ∀ (l : list var) (x : var) (sig1 : subst) (sig2 : subst) (s : term),
  ¬ (In x l) →
  (apply_subst (VAR x) (build_on_list_of_vars l s sig1 sig2)) ==
  (VAR x).
Proof.
intros. unfold not in H.
induction l.
- simpl. reflexivity.
- simpl. pose proof elt_not_in_list as H2. specialize (H2 x a l). unfold not in H2.
  specialize (H2 H). destruct H2.
  destruct (beq_nat a x) as []eqn:?.
  + symmetry in Heqb. apply beq_nat_eq in Heqb. symmetry in Heqb. specialize (H0
Heqb). destruct H0.
  + simpl in IHl. apply IHl. apply H1.
Qed.

Lemma lowenheim_rephrase2 :
  ∀ (t : term) (tau : subst) (x : var),
```

```
  (unifier t tau) →
  ¬ (In x (term_unique_vars t)) →
  (apply_subst (VAR x) (build_lowenheim_subst t tau)) ==
  (VAR x).
Proof.
intros. unfold build_lowenheim_subst. pose proof lowenheim_rephrase2_easy as H2.
specialize (H2 (term_unique_vars t) x (build_id_subst (term_unique_vars t)) tau t).
specialize (H2 H0). apply H2.
Qed.

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

```
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

3.4.a utilities

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

Lemma empty_subst_on_term:
 ∀ (t : term),
  apply_subst t [] == t.
Proof.
 intros. induction t.
 - reflexivity.
 - simpl. reflexivity.
 - simpl. reflexivity.
 - simpl. rewrite IHt1. rewrite IHt2. reflexivity.
 - simpl. rewrite IHt1. rewrite IHt2. reflexivity.
Qed.

Lemma app_subst_T0:
 ∀ (t : term),
 apply_subst t [] == T0 → t == T0.
Proof.
intros. rewrite empty_subst_on_term in H. apply H.
Qed.

Lemma T0_or_not_T0:
 ∀ (t : term),
 t == T0 ∨ ¬ (t == T0).
Proof.
 intros. pose proof classic. specialize (H (t == T0)). apply H.
Qed.

Lemma exists_subst:
 ∀ (t : term) (sig : subst),
 apply_subst t sig == T0 → ∃ s, apply_subst t s == T0.
Proof.
 intros. ∃ sig. apply H.
Qed.

Lemma t_id_eqv :
 ∀ (t : term),
 t == t.
Proof.
 intros. reflexivity.
Qed.
```

```
Lemma eq_some_eq_subst (s1 s2: subst) :
  (Some_subst s1 = Some_subst s2) → s1 = s2.
Proof.
  intros. congruence.
Qed.

Lemma None_is_not_Some (t: term):
  (find_unifier t) = None_subst → (∀ (sig: subst), ¬ (find_unifier t) = Some_subst sig).
Proof.
  intros.
  congruence.
Qed.

Lemma Some_is_not_None (sig: subst) (t: term):
  (find_unifier t) = Some_subst sig → ¬ (find_unifier t = None_subst).
Proof.
  intros.
  congruence.
Qed.

Lemma not_None_is_Some (t: term) :
  ¬ (find_unifier t = None_subst) → ∃ sig : subst, (find_unifier t) = Some_subst sig.
Proof.
  intros H.
  destruct (find_unifier t) as [ti | ].
  - ∃ ti. firstorder.
  - congruence.
Qed.

Lemma contrapositive_opposite :
  ∀ p q, (¬p → ¬q) → q →p.
Proof.
  intros.
  apply NNPP. firstorder.
Qed.

Lemma contrapositive :
∀ (p q : Prop), (p → q) → ( ¬q → ¬p).
Proof.
  intros.
  firstorder.
Qed.
```

3.4.b actual final proof extension

```
Lemma some_subst_unifiable:
 ∀ (t : term),
```

```
  (∃ sig, (find_unifier t) = Some_subst sig) → (unifiable t).
Proof.
 intros.
 destruct H as [sig1 H1].
 induction t.
 - unfold unifiable . ∃ []. unfold unifier. simpl. reflexivity.
 - simpl in H1. inversion H1.
 - unfold unifiable. ∃ sig1. unfold find_unifier in H1.
    remember (update_term (VAR v) (rec_subst (VAR v) (term_unique_vars (VAR v)) []))
in H1.
    destruct t.
    + unfold update_term in Heqt. pose proof simplify_eqv.
      specialize (H (apply_subst (VAR v) (rec_subst (VAR v) (term_unique_vars (VAR v))
[]))).
       symmetry in Heqt. apply eq_some_eq_subst in H1.
      rewrite H1 in H. rewrite H1 in Heqt.
     rewrite Heqt in H. symmetry in H. apply H.
    + simpl in H1. inversion H1.
    + inversion H1.
    + inversion H1.
    + inversion H1.
 - unfold unifiable. ∃ sig1. unfold find_unifier in H1.
   remember (update_term (t1 + t2) (rec_subst (t1 + t2) (term_unique_vars (t1 + t2)) []))
in H1.
  destruct t.
  + unfold update_term in Heqt. pose proof simplify_eqv.
    specialize (H (apply_subst (t1 + t2) (rec_subst (t1 + t2) (term_unique_vars (t1 + t2))
[]))).
       symmetry in Heqt. apply eq_some_eq_subst in H1.
      rewrite H1 in H. rewrite H1 in Heqt.
     rewrite Heqt in H. symmetry in H. apply H.
   + inversion H1.
   + inversion H1.
   + inversion H1.
   + inversion H1.
 - unfold unifiable. ∃ sig1. unfold find_unifier in H1.
   remember (update_term (t1 × t2) (rec_subst (t1 × t2) (term_unique_vars (t1 × t2)) []))
in H1.
  destruct t.
  + unfold update_term in Heqt. pose proof simplify_eqv.
    specialize (H (apply_subst (t1 × t2) (rec_subst (t1 × t2) (term_unique_vars (t1 ×
t2)) []))).
```

```
        symmetry in Heqt. apply eq_some_eq_subst in H1.
       rewrite H1 in H. rewrite H1 in Heqt.
      rewrite Heqt in H. symmetry in H. apply H.
    + inversion H1.
    + inversion H1.
    + inversion H1.
    + inversion H1.
Qed.

Lemma not_Some_is_None (t: term) :
 ( ¬ ∃ (sig : subst), (find_unifier t) = Some_subst sig) → (find_unifier t) = None_subst.
Proof.
  apply contrapositive_opposite.
  intros H.
  apply not_None_is_Some in H.
  tauto.
Qed.

Lemma not_unifiable_find_unifier_none_subst :
∀ (t : term),
   ¬ (unifiable t) → (find_unifier t) = None_subst.
Proof.
intros.
 pose proof some_subst_unifiable.
 specialize (H0 t).
 pose proof contrapositive.
 specialize (H1 ((∃ sig : subst, find_unifier t = Some_subst sig)) ((unifiable t))).
 specialize (H1 H0). specialize (H1 H).
 pose proof not_Some_is_None.
 specialize (H2 t H1).
 apply H2.
Qed.

Lemma Some_subst_unifiable :
∀ (t : term) (sig : subst),
   (find_unifier t) = Some_subst sig → (unifier t sig).
Proof.
intros.
 induction t.
 - simpl in H. apply eq_some_eq_subst in H. symmetry in H. rewrite H.
  unfold unifier. simpl. reflexivity.
 - simpl in H. inversion H.
 - unfold find_unifier in H. remember (update_term (VAR v) (rec_subst (VAR v) (term_unique_vars
(VAR v)) [])) in H.
```

```
    destruct t.
  + unfold update_term in Heqt. pose proof simplify_eqv.
      specialize (H0 (apply_subst (VAR v) (rec_subst (VAR v) (term_unique_vars (VAR v))
[]))).
        symmetry in Heqt. apply eq_some_eq_subst in H.
      rewrite H in H0. rewrite H in Heqt.
     rewrite Heqt in H0. symmetry in H0. apply H0.
  + inversion H.
  + inversion H.
  + inversion H.
  + inversion H.
 - unfold find_unifier in H. remember (update_term (t1 + t2) (rec_subst (t1 + t2) (term_unique_vars
(t1 + t2)) [])) in H.
    destruct t.
  + unfold update_term in Heqt. pose proof simplify_eqv.
      specialize (H0 (apply_subst (t1 + t2) (rec_subst (t1 + t2) (term_unique_vars (t1 +
t2)) []))).
       symmetry in Heqt. apply eq_some_eq_subst in H.
      rewrite H in H0. rewrite H in Heqt.
     rewrite Heqt in H0. symmetry in H0. apply H0.
  + inversion H.
  + inversion H.
  + inversion H.
  + inversion H.
 - unfold find_unifier in H. remember (update_term (t1 × t2) (rec_subst (t1 × t2) (term_unique_vars
(t1 × t2)) [])) in H.
    destruct t.
  + unfold update_term in Heqt. pose proof simplify_eqv.
      specialize (H0 (apply_subst (t1 × t2) (rec_subst (t1 × t2) (term_unique_vars (t1 ×
t2)) []))).
       symmetry in Heqt. apply eq_some_eq_subst in H.
      rewrite H in H0. rewrite H in Heqt.
     rewrite Heqt in H0. symmetry in H0. apply H0.
  + inversion H.
  + inversion H.
  + inversion H.
  + inversion H.
Qed.

Lemma unif_some_subst :
 ∀ (t: term),
 (∃ sig1, (unifier t sig1)) →
 (∃ sig2, (find_unifier t) = Some_subst sig2).
```

```
Proof.
 intros.
 destruct H as [sig1 H].
Admitted.

Lemma not_Some_not_unifiable (t: term) :
 ( ¬ ∃ (sig : subst), (find_unifier t) = Some_subst sig) → ¬ (unifiable t).
Proof.
 intros.
 pose proof not_Some_is_None.
 specialize (H0 t H).
 unfold unifiable.
 intro.
  unfold not in H.
 pose proof unif_some_subst.
 specialize (H2 t H1).
 specialize (H H2).
 apply H.
Qed.

Lemma unifiable_find_unifier_some_subst :
∀ (t : term),
   (unifiable t) → (∃ (sig : subst), (find_unifier t) = Some_subst sig).
Proof.
intros.
 pose proof contrapositive.
 specialize (H0 ( ¬ ∃ (sig : subst), (find_unifier t) = Some_subst sig) (¬ (unifiable t))).
 pose proof not_Some_not_unifiable.
 specialize (H1 t). specialize (H0 H1). apply NNPP in H0.
 - apply H0.
 - firstorder.
Qed.

Lemma find_unifier_is_unifier:
 ∀ (t : term),
  (unifiable t) → (unifier t (convert_to_subst (find_unifier t))).
Proof.
intros.
 pose proof unifiable_find_unifier_some_subst.
 specialize (H0 t H).
 unfold unifier. unfold unifiable in H. simpl. unfold convert_to_subst.
 destruct H0 as [sig H0]. rewrite H0.
 pose proof Some_subst_unifiable.
 specialize (H1 t sig). specialize (H1 H0).
```

```
 unfold unifier in H1.
 apply H1.
Qed.

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.
intros.
pose proof lowenheim_most_general_unifier as H1. pose proof find_unifier_is_unifier as H2.
specialize (H2 t H). specialize (H1 t (convert_to_subst (find_unifier t))).
specialize (H1 H2). unfold Lowenheim_Main. destruct (find_unifier t).
 - simpl. simpl in H1. apply H1.
 - simpl in H2. unfold unifier in H2. apply app_subst_T0 in H2. simpl.
   repeat simpl in H1. pose proof most_general_unifier_compat.
   specialize (H3 t T0 H2). specialize (H3 []).
   rewrite H3. unfold most_general_unifier. intros.
   unfold more_general_substitution. ∃ s'. unfold substitution_composition.
   intros. simpl. reflexivity.
Qed.

Lemma lowenheim_main_most_general_unifier:
 ∀ (t: term),
 ((unifiable t) → most_general_unifier t (convert_to_subst (Lowenheim_Main t)))
 ∧
 (~(unifiable t) → (Lowenheim_Main t) = None_subst ).
Proof.
 intros.
 split.
 - intros. apply builder_to_main.
  + apply H.
  + apply lowenheim_most_general_unifier. apply find_unifier_is_unifier. apply H.
 - intros. pose proof not_unifiable_find_unifier_none_subst.
   specialize (H0 t H). unfold Lowenheim_Main. rewrite H0. reflexivity.
Qed.
```

# Chapter 5

# Library B_Unification.list_util

```
Require Import List.
Import ListNotations.
Require Import Arith.
Import Nat.
Require Import Sorting.
Require Import Permutation.
Require Import Omega.
```

## 5.1 Introduction

The second half of the project revolves around the successive variable elimination algorithm for solving unification problems. While we could implement this algorithm with the same data structures used for Lowenheim's, this algorithm lends itself well to a new representation of terms as polynomials.

A polynomial is a list of monomials being added together, where a monomial is a list of variables being multiplied together. Since one of the rules is that 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. For simplicity when implementing and comparing these polynomials in Coq, we have opted to use the standard list structure, instead maintaining that the lists are maintained in our polynomial form after each stage.

In order to effectively implement polynomial lists in this way, a set of utilities are needed to allow us to easily perform operations on these lists. This file serves to implement and prove facts about these functions, as well as to expand upon the standard library when necessary.

## 5.2 Comparisons Between Lists

Checking if a list of natural numbers is sorted is easy enough. Comparing lists of lists of nats is slightly harder, and requires the use of a new function, called lex. lex simply takes in a comparison and applies the comparison across the list until it finds a point where the elements are not equal.

In all cases throughout this project, the comparator used will be the standard nat compare function.

For example, [1;2;3] is less than [1;2;4], and [1;2] is greater than [1].

```
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:

```
Lemma 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 $b > a$.

```
Lemma 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.
```

It is also useful to convert from the result of lex compare to a hypothesis about equality in Coq. Clearly, if lex compare returns Eq, the lists are exactly equal, and if it returns Lt or Gt they are not.

```
Lemma lex_eq : ∀ n m,
  lex compare n m = Eq ↔ n = m.
Proof.
  intros n. induction n; induction m; intros.
  - split; reflexivity.
  - split; intros; inversion H.
  - split; intros; inversion H.
  - split; intros; simpl in H.
    + destruct (a ?= a0) eqn:Hcomp; try inversion H. f_equal.
      × apply compare_eq_iff in Hcomp; auto.
      × apply IHn. auto.
    + inversion H. simpl. rewrite compare_refl.
      rewrite ← H2. apply IHn. reflexivity.
Qed.
```

```
Lemma lex_neq : ∀ n m,
  lex compare n m = Lt ∨ lex compare n m = Gt ↔ n ≠ m.
Proof.
  intros n. induction n; induction m.
  - simpl. split; intro. inversion H; inversion H0. contradiction.
  - simpl. split; intro. intro. inversion H0. auto.
  - simpl. split; intro. intro. inversion H0. auto.
  - clear IHm. split; intros.
    + destruct H; intro; apply lex_eq in H0; rewrite H in H0; inversion H0.
    + destruct (a ?= a0) eqn:Hcomp.
      × simpl. rewrite Hcomp. apply IHn. apply compare_eq_iff in Hcomp.
        rewrite Hcomp in H. intro. apply H. rewrite H0. reflexivity.
      × left. simpl. rewrite Hcomp. reflexivity.
      × right. simpl. rewrite Hcomp. reflexivity.
Qed.
```

```
Lemma lex_neq' : ∀ n m,
```

```
  (lex compare n m = Lt → n ≠ m) ∧
  (lex compare n m = Gt → n ≠ m).
Proof.
  intros n m. split.
  - intros. apply lex_neq. auto.
  - intros. apply lex_neq. auto.
Qed.
```

It is also useful to be able to flip the arguments of a call to lex compare, since these two comparisons impact each other directly.

If lex returns that $n = m$, then this also means that $m = n$. More interesting is that if $n < m$, then $m > n$.

```
Lemma lex_rev_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_rev_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.

```
Lemma 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.
```

## 5.3 Extensions to the Standard Library

There were some facts about the standard library list functions that we found useful to prove, as they repeatedly came up in proofs of our more complex custom list functions.

Specifically, because we are comparing sorted lists, it is often easier to disregard the sortedness of the lists and instead compare them as Permutations of one another. As a result, many of the lemmas in the rest of this file revolve around proving that two lists are Permutations of one another.

### 5.3.1 Facts about In

First, a very simple fact about In. This mostly follows from the standard library lemma Permutation_in, but is more convenient for some of our proofs when formalized like this.

```
Lemma Permutation_not_In : ∀ (A:Type) a (l l':list A),
  Permutation l l' →
  ¬ In a l →
  ¬ In a l'.
Proof.
  intros A a l l' H H0. intro. apply H0. apply Permutation_sym in H.
  apply (Permutation_in a) in H; auto.
Qed.
```

Something else that seems simple but proves very useful to know is that if there are no elements In a list, that list must be empty.

```
Lemma nothing_in_empty : ∀ {A} (l:list A),
  (∀ a, ¬ In a l) →
  l = [].
Proof.
  intros A l H. destruct l; auto. pose (H a). simpl in n. exfalso.
  apply n. auto.
Qed.
```

### 5.3.2 Facts about incl

Next are some useful lemmas about incl. First is that if one list is included in another, but one element of the second list is not in the first, then the first list is still included in the second with that element removed.

```
Lemma incl_not_in : ∀ A a (l m : list A),
  incl l (a :: m) →
  ¬ In a l →
  incl l m.
Proof.
  intros A a l m Hincl Hnin. unfold incl in *. intros a0 Hin.
  simpl in Hincl. destruct (Hincl a0); auto. rewrite H in Hnin. contradiction.
Qed.
```

We also found it useful to relate Permutation to incl; if two lists are permutations of each other, then they must be set equivalent, or contain all of the same elements.

```
Lemma Permutation_incl : ∀ {A} (l m : list A),
  Permutation l m → incl l m ∧ incl m l.
Proof.
  intros A l m H. apply Permutation_sym in H as H0. split.
  + unfold incl. intros a. apply (Permutation_in _ H).
  + unfold incl. intros a. apply (Permutation_in _ H0).
Qed.
```

Unfortunately, the definition above cannot be changed into an iff relation, as incl proves nothing about the counts in the lists. We can, however, prove that if some $m$ includes all the elements of a list, then it also includes all the elements of all permutations of that list.

```
Lemma incl_Permutation : ∀ {A:Type} (l l' m:list A),
  Permutation l l' →
  incl l m →
  incl l' m.
Proof.
  intros A l l' m H H0. apply Permutation_incl in H as [].
  apply incl_tran with (m:=l); auto.
Qed.
```

A really simple lemma is that if some $l$ is included in the empty list, then that list must also be empty.

```
Lemma incl_nil : ∀ {X:Type} (l:list X),
  incl l [] ↔ l = [].
Proof.
  intros X l. unfold incl. split; intro H.
  - destruct l; [auto | destruct (H x); intuition].
  - intros a Hin. destruct l; [auto | rewrite H in Hin; auto].
Qed.
```

The last fact about incl is simply a new way of formalizing the definition that is convenient for some proofs.

```
Lemma incl_cons_inv : ∀ (A:Type) (a:A) (l m : list A),
  incl (a :: l) m → In a m ∧ incl l m.
Proof.
  intros A a l m H. split.
  - unfold incl in H. apply H. intuition.
  - unfold incl in *. intros b Hin. apply H. intuition.
Qed.
```

### 5.3.3 Facts about count_occ

Next is some facts about count_occ. Firstly, if two lists are permutations of each other, than every element in the first list has the same number of occurences in the second list.

```
Lemma count_occ_Permutation : ∀ (A:Type) Aeq_dec a (l l':list A),
  Permutation l l' →
  count_occ Aeq_dec l a = count_occ Aeq_dec l' a.
Proof.
  intros A Aeq_dec a l l' H. induction H.
  - auto.
  - simpl. destruct (Aeq_dec x a); auto.
  - simpl. destruct (Aeq_dec y a); destruct (Aeq_dec x a); auto.
  - rewrite ← IHPermutation2. rewrite IHPermutation1. auto.
Qed.
```

count_occ also distributes over app, instead becoming addition, which is useful especially when dealing with count occurences of concatenated lists during induction.

```
Lemma count_occ_app : ∀ (A:Type) a (l m:list A) Aeq_dec,
  count_occ Aeq_dec (l++m) a = add (count_occ Aeq_dec l a) (count_occ Aeq_dec m a).
Proof.
  intros A a l m Aeq_dec. induction l.
  - simpl. auto.
  - simpl. destruct (Aeq_dec a0 a); simpl; auto.
Qed.
```

It is also convenient to reason about the relation between count_occ and remove. If the element being removed is the same as the one being counted, then the count is obviously 0; if the elements are different, then the count is the same with or without the remove.

```
Lemma count_occ_remove : ∀ {A} Aeq_dec (a:A) p,
  count_occ Aeq_dec (remove Aeq_dec a p) a = 0.
Proof.
  intros A Aeq_dec a p. induction p.
  - simpl. auto.
  - simpl. destruct (Aeq_dec a a0) eqn:Haa0.
    + apply IHp.
    + simpl. destruct (Aeq_dec a0 a); try (symmetry in e; contradiction).
      apply IHp.
Qed.
```

```
Lemma count_occ_neq_remove : ∀ {A} Aeq_dec (a:A) b p,
  a ≠ b →
  count_occ Aeq_dec (remove Aeq_dec a p) b =
  count_occ Aeq_dec p b.
Proof.
  intros A Aeq_dec a b p H. induction p; simpl; auto. destruct (Aeq_dec a a0).
  - destruct (Aeq_dec a0 b).
    + rewrite ← e0 in H. rewrite e in H. contradiction.
    + apply IHp.
```

```
  - simpl. destruct (Aeq_dec a0 b); auto.
Qed.
```

### 5.3.4 Facts about concat

Similarly to the lemma Permutation_map, Permutation_concat shows that if two lists are permutations of each other then the concatenation of each list are also permutations.

```
Lemma Permutation_concat : ∀ {A} (l m:list (list A)),
  Permutation l m →
  Permutation (concat l) (concat m).
Proof.
  intros A l m H. induction H.
  - auto.
  - simpl. apply Permutation_app_head. auto.
  - simpl. apply Permutation_trans with (l':=(concat l ++ y ++ x)).
    + rewrite app_assoc. apply Permutation_app_comm.
    + apply Permutation_trans with (l':=(concat l ++ x ++ y)).
      × apply Permutation_app_head. apply Permutation_app_comm.
      × rewrite (app_assoc x y). apply Permutation_app_comm.
  - apply Permutation_trans with (l':=(concat l')); auto.
Qed.
```

Before the creation of this lemma, it was relatively hard to reason about whether elements are in the concatenation of a list of lists. This lemma states that if there is a list in the list of lists that contains the desired element, then that element will be in the concatenated version.

```
Lemma In_concat_exists : ∀ (A:Type) ll (a:A),
  (∃ l, In l ll ∧ In a l) ↔ In a (concat ll).
Proof.
  intros A ll a. split; intros H.
  - destruct H as [l[]]. apply In_split in H. destruct H as [l1[l2 H]].
    rewrite H. apply Permutation_in with (l:=(concat (l :: l1 ++ l2))).
    + apply Permutation_concat. apply Permutation_middle.
    + simpl. apply in_app_iff. auto.
  - induction ll.
    + inversion H.
    + simpl in H. apply in_app_iff in H. destruct H.
      × ∃ a0. split; intuition.
      × destruct IHll; auto. ∃ x. intuition.
Qed.
```

This particular lemma is useful if the function being mapped returns a list of its input type. If the resulting lists are concatenated after, then the result is the same as mapping the function without converting the output to lists.

```
Lemma concat_map : ∀ {A B:Type} (f:A→B) (l:list A),
  concat (map (fun a ⇒ [f a]) l) = map f l.
Proof.
  intros A B f l. induction l.
  - auto.
  - simpl. f_equal. apply IHl.
Qed.
```

Another fact similar to the last is that if you concatenate the result of mapping a function that maps a function over a list, we can rearrange the order of the concat and the maps.

```
Lemma concat_map_map : ∀ A B C l (f:B→C) (g:A→list B),
  concat (map (fun a ⇒ map f (g a)) l) =
  map f (concat (map g l)).
Proof.
  intros. induction l; auto.
  simpl. rewrite map_app. f_equal. auto.
Qed.
```

Lastly, if you map a function that converts every element of a list to nil, and then concat the list of nils, you end with nil.

```
Lemma concat_map_nil : ∀ {A} (p:list A),
  concat (map (fun x ⇒ []) p) = (@nil A).
Proof.
  induction p; auto.
Qed.
```

### 5.3.5 Facts about Forall and existsb

This is similar to the inverse of Forall; any element in the list must hold the specified relation if Forall *Rel* is true of the list.

```
Lemma Forall_In : ∀ (A:Type) (l:list A) a Rel,
  In a l → Forall Rel l → Rel a.
Proof.
  intros A l a Rel Hin Hfor. apply (Forall_forall Rel l); auto.
Qed.
```

In Coq, existsb is effectively the "or" to Forall's "and" when reasoning about lists. If there does not exist a single element in the list where $f$ is true, then $(f\ a)$ must be false for all elements of the list.

```
Lemma existsb_false_forall : ∀ {A} f (l:list A),
  existsb f l = false →
  (∀ a, In a l → (f a) = false).
Proof.
  intros A f l H a Hin. destruct (f a) eqn:Hfa.
```

```
  - exfalso. rewrite <- Bool.negb_true_iff in H. apply (Bool.eq_true_false_abs _ H).
    rewrite Bool.negb_false_iff. apply existsb_exists. ∃ a. split; auto.
  - auto.
Qed.
```

Similarly to Forall_In, this lemma is just another way of formalizing the definition of Forall that proves useful when dealing with **StronglySorted** lists.

```
Lemma Forall_cons_iff : ∀ (A:Type) Rel a (l:list A),
  Forall Rel (a::l) <-> Forall Rel l ∧ Rel a.
Proof.
  intros A Rel a l. split.
  - intro H. split.
    + rewrite Forall_forall in H. apply Forall_forall. intros x Hin.
      apply H. intuition.
    + apply Forall_inv in H. auto.
  - intros []. apply Forall_cons; auto.
Qed.
```

If a relation holds for all elements of a list l, then the relation still holds if some elements are removed from the list.

```
Lemma Forall_remove : ∀ (A:Type) Aeq_dec Rel a (l:list A),
  Forall Rel l -> Forall Rel (remove Aeq_dec a l).
Proof.
  intros A Aeq_dec Rel a l H. induction l.
  - simpl. auto.
  - simpl. apply Forall_cons_iff in H. destruct (Aeq_dec a a0).
    + apply IHl. apply H.
    + apply Forall_cons_iff. split.
      × apply IHl. apply H.
      × apply H.
Qed.
```

This next lemma is particularly useful for relating **StronglySorted** lists to Sorted lists; if some comparator holds for all elements of $p$, then this can be converted to the **HdRel** proposition used by Sorted.

```
Lemma Forall_HdRel : ∀ {X:Type} c a (p:list X),
  Forall (c a) p -> HdRel c a p.
Proof.
  intros X c a p H. destruct p.
  - apply HdRel_nil.
  - apply HdRel_cons. apply Forall_inv in H. auto.
Qed.
```

Lastly, if some property $(c\ a)$ is true for all elements in a list $p$, and the elements of a second list $g$ are all included in $p$, then the property is also true for the elements in $g$.

```
Lemma Forall_incl : ∀ {X:Type} (c:X→X→Prop) a (p g:list X),
  Forall (c a) p → incl g p → Forall (c a) g.
Proof.
  intros X c a p g H H0. induction g.
  - apply Forall_nil.
  - rewrite Forall_forall in H. apply Forall_forall. intros x Hin.
    apply H. unfold incl in H0. apply H0. intuition.
Qed.
```

### 5.3.6 Facts about remove

There are surprisingly few lemmas about remove in the standard library, so in addition to those proven in other places, we opted to add quite a few simple facts about remove. First is that if an element is in a list after something has been removed, then clearly it was in the list before as well.

```
Lemma In_remove : ∀ {A:Type} Aeq_dec a b (l:list A),
  In a (remove Aeq_dec b l) → In a l.
Proof.
  intros A Aeq_dec a b l H. induction l as [|c l IHl].
  - contradiction.
  - destruct (Aeq_dec b c) eqn:Heq; simpl in H; rewrite Heq in H.
    + right. auto.
    + destruct H; [rewrite H; intuition | right; auto].
Qed.
```

Similarly to Forall_remove, if a list was StronglySorted before something was removed then it is also StronglySorted after.

```
Lemma StronglySorted_remove : ∀ {A:Type} Aeq_dec Rel a (l:list A),
  StronglySorted Rel l → StronglySorted Rel (remove Aeq_dec a l).
Proof.
  intros A Aeq_dec Rel a l H. induction l.
  - simpl. auto.
  - simpl. apply StronglySorted_inv in H. destruct (Aeq_dec a a0).
    + apply IHl. apply H.
    + apply SSorted_cons.
      × apply IHl. apply H.
      × apply Forall_remove. apply H.
Qed.
```

If the item being removed from a list isn't in the list, then the list is equal with or without the remove.

```
Lemma not_In_remove : ∀ (A:Type) Aeq_dec a (l : list A),
  ¬ In a l → (remove Aeq_dec a l) = l.
```

```
Proof.
  intros A Aeq_dec a l H. induction l.
  - simpl. reflexivity.
  - simpl. destruct (Aeq_dec a a0).
    + simpl. rewrite e in H. exfalso. apply H. intuition.
    + rewrite IHl. reflexivity. intro Hin. apply H. intuition.
Qed.
```

remove also distributes over append.

```
Lemma remove_distr_app : ∀ (A:Type) Aeq_dec x (l l':list A),
  remove Aeq_dec x (l ++ l') = remove Aeq_dec x l ++ remove Aeq_dec x l'.
Proof.
  intros A Aeq_dec x l l'. induction l; intros.
  - simpl. auto.
  - simpl. destruct (Aeq_dec x a).
    + apply IHl.
    + simpl. f_equal. apply IHl.
Qed.
```

More interestingly, if two lists were permutations before, they are also permutations after the same element has been removed from both lists.

```
Lemma remove_Permutation : ∀ (A:Type) Aeq_dec a (l l':list A),
  Permutation l l' →
  Permutation (remove Aeq_dec a l) (remove Aeq_dec a l').
Proof.
  intros A Aeq_dec a l l' H. induction H.
  - auto.
  - simpl. destruct (Aeq_dec a x); auto.
  - simpl. destruct (Aeq_dec a y); destruct (Aeq_dec a x); auto.
    apply perm_swap.
  - apply Permutation_trans with (l':=(remove Aeq_dec a l')); auto.
Qed.
```

remove is also commutative with itself.

```
Lemma remove_remove : ∀ {A:Type} Aeq_dec (a b:A) p,
  remove Aeq_dec a (remove Aeq_dec b p) =
  remove Aeq_dec b (remove Aeq_dec a p).
Proof.
  intros A Aeq_dec a b p. induction p as [|c]; simpl; auto.
  destruct (Aeq_dec a b); destruct (Aeq_dec b c); destruct (Aeq_dec a c).
  - auto.
  - rewrite ← e0 in n. rewrite e in n. contradiction.
  - rewrite ← e in n. rewrite e0 in n. contradiction.
  - simpl. destruct (Aeq_dec a c); try contradiction.
```

```
  destruct (Aeq_dec b c); try contradiction. rewrite IHp. auto.
- rewrite e in n. rewrite e0 in n. contradiction.
- simpl. destruct (Aeq_dec b c); try contradiction. auto.
- simpl. destruct (Aeq_dec a c); try contradiction. auto.
- simpl. destruct (Aeq_dec a c); try contradiction.
  destruct (Aeq_dec b c); try contradiction. rewrite IHp. auto.
Qed.
```

Lastly, if an element is being removed from a particular list twice, the inner remove is redundant and can be removed.

```
Lemma remove_pointless : ∀ {A Aeq_dec} (a:A) p q,
  remove Aeq_dec a (remove Aeq_dec a p ++ q) =
  remove Aeq_dec a (p ++ q).
Proof.
  intros A Aeq_dec a p q. induction p; auto. simpl. destruct (Aeq_dec a a0) eqn:Heq.
  - apply IHp.
  - simpl. rewrite Heq. f_equal. apply IHp.
Qed.
```

### 5.3.7 Facts about nodup and NoDup

Next up - the NoDup proposition and the closely related nodup function. The first lemma states that if there are no duplicates in a list, then two items in that list must not be equal.

```
Lemma NoDup_neq : ∀ {X:Type} (m : list X) a b,
  NoDup (a :: b :: m) →
  a ≠ b.
Proof.
  intros X m a b Hdup. apply NoDup_cons_iff in Hdup as [].
  apply NoDup_cons_iff in H0 as []. intro. apply H. simpl. auto.
Qed.
```

In a similar vein as many of the other remove lemmas, if there were no duplicates in a list before the remove then there are still none after.

```
Lemma NoDup_remove : ∀ (A:Type) Aeq_dec a (l:list A),
  NoDup l → NoDup (remove Aeq_dec a l).
Proof.
  intros A Aeq_dec a l H. induction l.
  - simpl. auto.
  - simpl. destruct (Aeq_dec a a0).
    + apply IHl. apply NoDup_cons_iff in H. intuition.
    + apply NoDup_cons.
      × apply NoDup_cons_iff in H as []. intro. apply H.
        apply (In_remove Aeq_dec a0 a l H1).
```

```
      × apply IHl. apply NoDup_cons_iff in H; intuition.
Qed.
```

Another lemma similar to NoDup_neq is NoDup_forall_neq; if every element in a list is not equal to a certain $a$, and the list has no duplicates as is, then it is safe to add $a$ to the list without creating duplicates.

```
Lemma NoDup_forall_neq : ∀ (A:Type) a (l:list A),
  Forall (fun b ⇒ a ≠ b) l →
  NoDup l →
  NoDup (a :: l).
Proof.
  intros A a l Hf Hn. apply NoDup_cons.
  - intro. induction l.
    + inversion H.
    + apply Forall_cons_iff in Hf as []. apply IHl.
      × apply H0.
      × apply NoDup_cons_iff in Hn. apply Hn.
      × simpl in H. destruct H; auto. rewrite H in H1. contradiction.
  - auto.
Qed.
```

This lemma is really just a reformalization of NoDup_remove_2, which allows us to easily prove that some $x$ is not in the preceeding elements $l1$ or the following elements $l2$ when the whole list $l$ has no duplicates.

```
Lemma NoDup_In_split : ∀ {A:Type} (x:A) l l1 l2,
  l = l1 ++ x :: l2 →
  NoDup l →
  ¬ In x l1 ∧ ¬ In x l2.
Proof.
  intros A x l l1 l2 H H0. rewrite H in H0.
  apply NoDup_remove_2 in H0. split; intro; intuition.
Qed.
```

Now some facts about the function nodup; if the **NoDup** predicate is already true about a certain list, then calling nodup on it changes nothing.

```
Lemma no_nodup_NoDup : ∀ (A:Type) Aeq_dec (l:list A),
  NoDup l →
  nodup Aeq_dec l = l.
Proof.
  intros A Aeq_dec l H. induction l.
  - auto.
  - simpl. apply NoDup_cons_iff in H as []. destruct (in_dec Aeq_dec a l).
    contradiction. f_equal. auto.
Qed.
```

If a list is sorted (with a transitive relation) before calling nodup on it, the list is also sorted after.

```
Lemma Sorted_nodup : ∀ (A:Type) Aeq_dec Rel (l:list A),
  Relations_1.Transitive Rel →
  Sorted Rel l →
  Sorted Rel (nodup Aeq_dec l).
Proof.
  intros A Aeq_dec Rel l Ht H. apply Sorted_StronglySorted in H; auto.
  apply StronglySorted_Sorted. induction l.
  - auto.
  - simpl. apply StronglySorted_inv in H as []. destruct (in_dec Aeq_dec a l).
    + apply IHl. apply H.
    + apply SSorted_cons.
      × apply IHl. apply H.
      × rewrite Forall_forall in H0. apply Forall_forall. intros x Hin.
        apply H0. apply nodup_In in Hin. auto.
Qed.
```

We can also show that in some cases, if there are repeated calls to nodup, they are "pointless" - in other words, we can remove the inner call and only keep the outer one.

```
Lemma nodup_pointless : ∀ m a,
  nodup Nat.eq_dec (m ++ nodup Nat.eq_dec a) = nodup Nat.eq_dec (m ++ a).
Proof.
  intros m a. induction m.
  - simpl. rewrite no_nodup_NoDup; auto. apply NoDup_nodup.
  - simpl. destruct in_dec; destruct in_dec.
    + auto.
    + exfalso. apply n. apply in_app_iff in i; destruct i. intuition.
      apply nodup_In in H; intuition.
    + exfalso. apply n. apply in_app_iff in i; destruct i; intuition.
      apply in_app_iff. right. apply nodup_In; auto.
    + f_equal. auto.
Qed.
```

And lastly, similarly to our other Permutation lemmas this far, if two lists were permutations of each other before nodup they are also permutations after.

This lemma was slightly more complex than previous Permutation lemmas, but the proof is still very similar. It is solved by induction on the Permutation hypothesis. The first and last cases are trivial, and the second case (where we must prove Permutation $(x::l)$ $(x::l')$) becomes simple with the use of Permutation_in.

The last case (where we must show Permutation $(x::y::l)$ $(y::x::l)$) was slightly complicated by the fact that destructing in_dec gives us a hypothesis like In $x$ $(y::l)$, which seems useless in reasoning about the other list at first. However, by also destructing whether or not x and y are equal, we can easily prove this case as well

```
Lemma Permutation_nodup : ∀ A Aeq_dec (l m:list A),
  Permutation l m → Permutation (nodup Aeq_dec l) (nodup Aeq_dec m).
Proof.
  intros. induction H.
  - auto.
  - simpl. destruct (in_dec Aeq_dec x l).
    + apply Permutation_in with (l':=l') in i; auto. destruct in_dec; try contradiction.
      auto.
    + assert (¬ In x l'). intro. apply n. apply Permutation_in with (l':=l) in H0; auto.
      apply Permutation_sym; auto. destruct in_dec; try contradiction.
      apply perm_skip. auto.
  - destruct (in_dec Aeq_dec y (x::l)). destruct i.
    + rewrite H. simpl. destruct (Aeq_dec y y); try contradiction. destruct in_dec.
      auto. apply perm_skip. auto.
    + simpl. destruct (Aeq_dec x y). destruct in_dec; destruct (Aeq_dec y x);
      try (symmetry in e; contradiction). rewrite e in i. destruct in_dec; try contradiction.
      auto. assert (¬ In y l). intro; apply n; rewrite e; auto.
      destruct in_dec; try contradiction. destruct in_dec; try contradiction.
      destruct in_dec; destruct (Aeq_dec y x); try (symmetry in e; contradiction).
      auto. apply perm_skip. auto.
    + simpl. destruct (Aeq_dec x y). destruct in_dec. destruct (Aeq_dec y x);
      try (symmetry in e; contradiction). rewrite e0. destruct in_dec; try contradiction.
      auto. destruct (Aeq_dec y x); try (symmetry in e; contradiction).
      assert (¬ In y l). intro; apply n0; rewrite e; auto. destruct in_dec; try
contradiction.
      rewrite e0. apply perm_skip; auto. assert (¬ In y l). intro; apply n; intuition.
      destruct in_dec; try contradiction. destruct in_dec; destruct (Aeq_dec y x);
      try (symmetry in e; contradiction). auto. apply perm_swap.
  - apply Permutation_trans with (l':=(nodup Aeq_dec l')); auto.
Qed.
```

### 5.3.8 Facts about partition

The final function in the standard library we found it useful to prove facts about is partition. First, we show the relation between partition and filter: filtering a list gives you a result that is equal to the first list partition would return. This lemma is proven one way, and then reformalized to be more useful in later proofs.

```
Lemma partition_filter_fst {X} p l :
  fst (partition p l) = @filter X p l.
Proof.
  induction l; simpl.
```

```
  - trivial.
  - rewrite ← IHl.
    destruct (partition p l); simpl.
    destruct (p a); now simpl.
Qed.
```

```
Lemma partition_filter_fst' : ∀ {X} p (l t f : list X),
     partition p l = (t, f) →
     t = @filter X p l .
Proof.
  intros X p l t f H.
  rewrite ← partition_filter_fst.
  now rewrite H.
Qed.
```

We would like to be able to state a similar fact about the second list returned by partition, but clearly these are all the elements "thrown out" by filter. Instead, we first create a simple definition for negating a function, and prove two quick facts about the relation between some $p$ and neg $p$.

```
Definition neg {X:Type} := fun (f:X→bool) ⇒ fun (a:X) ⇒ (negb (f a)).
```

```
Lemma neg_true_false : ∀ {X} (p:X→bool) (a:X),
  (p a) = true ↔ neg p a = false.
Proof.
  intros X p a. unfold neg. split; intro.
  - rewrite H. auto.
  - destruct (p a); intuition.
Qed.
```

```
Lemma neg_false_true : ∀ {X} (p:X→bool) (a:X),
  (p a) = false ↔ neg p a = true.
Proof.
  intros X p a. unfold neg. split; intro.
  - rewrite H. auto.
  - destruct (p a); intuition.
Qed.
```

With the addition of this neg proposition, we can now prove two lemmas relating the second partition list and filter in the same way we proved the lemmas about the first partition list.

```
Lemma partition_filter_snd {X} p l :
  snd (partition p l) = @filter X (neg p) l.
Proof.
  induction l; simpl.
  - reflexivity.
  - rewrite ← IHl.
```

```
    destruct (partition p l); simpl.
    destruct (p a) eqn:Hp.
    + simpl. apply neg_true_false in Hp. rewrite Hp; auto.
    + simpl. apply neg_false_true in Hp. rewrite Hp; auto.
Qed.
Lemma partition_filter_snd' : ∀ {X} p (l t f : list X),
  partition p l = (t, f) →
  f = @filter X (neg p) l.
Proof.
  intros X p l t f H.
  rewrite ← partition_filter_snd.
  now rewrite H.
Qed.
```

These lemmas about partition and filter are now put to use in two important lemmas about partition. If some list $l$ is partitioned into two lists (t, $f$), then every element in $t$ must return true for the filtering predicate and every element in $f$ must return false.

```
Lemma part_fst_true : ∀ X p (l t f : list X),
  partition p l = (t, f) →
  (∀ a, In a t → p a = true).
Proof.
  intros X p l t f Hpart a Hin.
  assert (Hf: t = filter p l).
  - now apply partition_filter_fst' with f.
  - assert (Hass := filter_In p a l).
    apply Hass.
    now rewrite ← Hf.
Qed.
Lemma part_snd_false : ∀ X p (x t f : list X),
  partition p x = (t, f) →
  (∀ a, In a f → p a = false).
Proof.
  intros X p l t f Hpart a Hin.
  assert (Hf: f = filter (neg p) l).
  - now apply partition_filter_snd' with t.
  - assert (Hass := filter_In (neg p) a l).
    rewrite ← neg_false_true in Hass.
    apply Hass.
    now rewrite ← Hf.
Qed.
```

Next is a rather obvious but useful lemma, which states that if a list $p$ was split into ($l$, $r$) then appending these lists back together results in a list that is a permutation of the original.

```
Lemma partition_Permutation : ∀ {A:Type} f (p l r: list A),
  partition f p = (l, r) →
  Permutation p (l++r).
Proof.
  intros A f p. induction p; intros.
  - simpl in H. inversion H. auto.
  - simpl in H. destruct (partition f p). destruct (f a); inversion H.
    + simpl. apply perm_skip. apply IHp. f_equal. auto.
    + apply Permutation_trans with (l':=(a::l1 ++ l)). apply perm_skip.
      apply Permutation_trans with (l':=(l++l1)). apply IHp. f_equal.
      auto. apply Permutation_app_comm. apply Permutation_app_comm with (l:=(a::l1)).
Qed.
```

The last and hardest fact about partition states that if the list being partitioned was already sorted, then the resulting two lists will also be sorted. This seems simple, as partition iterates through the elements in order and maintains the order in its children, but was surprisingly difficult to prove.

After performing induction, the next step was to destruct $(f\ a)$, to see which of the two lists the induction element would end up in. In both cases, the list that *doesn't* receive the new element is already clearly sorted by the induction hypothesis, but proving the other one is sorted is slightly harder.

By using Forall_HdRel (defined earlier), we reduced the problem in both cases to only having to show that the new element holds the relation $c$ on all elements of the list it was consed onto. After some manipulation and the use of partition_Permutation and Forall_incl, this follows from the fact that we know the new element holds the relation on all elements of the original list $p$, and therefore also holds it on the elements of the partitioned list.

```
Lemma part_Sorted : ∀ {X:Type} (c:X→X→Prop) f p,
  Relations_1.Transitive c →
  Sorted c p →
  ∀ l r, partition f p = (l, r) →
  Sorted c l ∧ Sorted c r.
Proof.
  intros X c f p Htran Hsort. induction p; intros.
  - simpl in H. inversion H. auto.
  - assert (H0:=H); auto. simpl in H. destruct (partition f p) as [g d].
    destruct (f a); inversion H.
    + assert (Forall (c a) g ∧ Sorted c g ∧ Sorted c r → Sorted c (a::g) ∧ Sorted c
r).
      × intros H4. split. apply Sorted_cons. apply H4. apply Forall_HdRel. apply H4.
apply H4.
      × apply H1. split.
        – apply Sorted_StronglySorted in Hsort; auto.
```

```
        apply StronglySorted_inv in Hsort as []. apply (Forall_incl _ _ _ _ H5).
        apply partition_Permutation in H0. rewrite <- H2 in H0. simpl in H0.
        apply Permutation_cons_inv in H0. apply Permutation_incl in H0 as [].
        unfold incl. unfold incl in H6. intros a0 Hin. apply H6. intuition.
      - apply IHp. apply Sorted_inv in Hsort; apply Hsort. f_equal. auto.
  + assert (Forall (c a) d /\ Sorted c l /\ Sorted c d -> Sorted c l /\ Sorted c (a::d)).
    * intros H4. split. apply H4. apply Sorted_cons. apply H4. apply Forall_HdRel.
apply H4.
    * apply H1. split.
      - apply Sorted_StronglySorted in Hsort; auto.
        apply StronglySorted_inv in Hsort as []. apply (Forall_incl _ _ _ _ H5).
        apply partition_Permutation in H0. rewrite <- H3 in H0. simpl in H0.
        apply Permutation_trans with (l'':=(a::d++l)) in H0.
        apply Permutation_cons_inv in H0.
        apply Permutation_trans with (l'':=(l++d)) in H0.
        apply Permutation_incl in H0 as []. unfold incl. unfold incl in H6.
        intros a0 Hin. apply H6. intuition. apply Permutation_app_comm.
        apply Permutation_app_comm with (l':=(a::d)).
      - apply IHp. apply Sorted_inv in Hsort; apply Hsort. f_equal. auto.
Qed.
```

## 5.4 New Functions over Lists

In order to easily perform the operations we need on lists, we defined three major list functions of our own, each with their own proofs. These generalized list functions all help to make it much easier to deal with our polynomial and monomial lists later in the development.

### 5.4.1 Distributing two Lists: distribute

The first and most basic of the three is distribute. Similarly to the "FOIL" technique learned in middle school for multiplying two polynomials, this function serves to create every combination of one element from each list. It is done concisely with the use of higher order functions below.

```
Definition distribute {A} (l m : list (list A)) : list (list A) :=
  concat (map (fun a:(list A) => (map (app a) l)) m).
```

The distribute function will play a larger role later, mostly as a part of our polynomial multiplication function. For now, however, there are only two very simple lemmas to be proven, both stating that distributing nil over a list results in nil.

```
Lemma distribute_nil : forall {A:Type} (p:list (list A)),
  distribute [] p = [].
Proof.
```

```
  intros A p. induction p.
  - auto.
  - unfold distribute in *. simpl in *. auto.
Qed.

Lemma distribute_nil_r : ∀ {A:Type} (p:list (list A)),
  distribute p [] = [].
Proof.
  intros A p. induction p.
  - auto.
  - unfold distribute in *. simpl in *. auto.
Qed.
```

### 5.4.2 Cancelling out Repeated Elements: nodup_cancel

The next list function, and possibly the most prolific function in our entire development, is nodup_cancel. Similarly to the standard library nodup function, nodup_cancel takes a list that may or may not have duplicates in it and returns a list without duplicates.

The difference between ours and the standard function is that rather than just removing all duplicates and leaving one of each element, the elements in a nodup_cancel list cancel out in pairs. For example, the list [1;1;1] would become [1], whereas [1;1;1;1] would become [].

This is implemented with the count_occ function and remove, and is largely the reason for needing so many lemmas about those two functions. If there is an *even* number of occurences of an element $a$ in the original list ($a$::$l$), which implies there is an *odd* number of occurences of this element in $l$, then all instances are removed. On the other hand, if there is an *odd* number of occurences in the original list, one occurence is kept, and the rest are removed.

By calling nodup_cancel recursively on *xs* *before* calling remove, Coq is easily able to determine that *xs* is the decreasing argument, removing the need for a more complicated definition with "fuel".

```
Fixpoint nodup_cancel {A} Aeq_dec (l : list A) : list A :=
  match l with
  | [] ⇒ []
  | x::xs ⇒
    let count := (count_occ Aeq_dec xs x) in
    let xs' := (remove Aeq_dec x (nodup_cancel Aeq_dec xs)) in
    if (even count) then x::xs' else xs'
  end.
```

Now onto lemmas. To begin with, there are a few facts true of nodup that are also true of nodup_cancel, which are useful in many proofs. nodup_cancel_in is the same as the standard library's *nodup_in*, with one important difference: this implication is *not* bidirectional. Because even parity elements are removed completely, not all elements in $l$ are guaranteed to be in nodup_cancel $l$.

NoDup_nodup_cancel is much simpler, and effectively exactly the same as NoDup_nodup.

In these proofs, and most others from this point on, the shape will be very similar to the proof of the corresponding nodup proof. The main difference is that, instead of destructing in_dec like one would for nodup, we destruct the evenness of count_occ, as that is what drives the main if statement of the function.

```
Lemma nodup_cancel_in : ∀ (A:Type) Aeq_dec a (l:list A),
  In a (nodup_cancel Aeq_dec l) → In a l.
Proof.
  intros A Aeq_dec a l H. induction l as [|b l IHl].
  - contradiction.
  - simpl in H. destruct (Aeq_dec a b).
    + rewrite e. intuition.
    + right. apply IHl. destruct (even (count_occ Aeq_dec l b)).
      × simpl in H. destruct H. rewrite H in n. contradiction.
        apply In_remove in H. auto.
      × apply In_remove in H. auto.
Qed.
```

```
Lemma NoDup_nodup_cancel : ∀ (A:Type) Aeq_dec (l:list A),
NoDup (nodup_cancel Aeq_dec l).
Proof.
  induction l as [|a l' Hrec]; simpl.
  - constructor.
  - destruct (even (count_occ Aeq_dec l' a)); simpl.
    + apply NoDup_cons; [apply remove_In | apply NoDup_remove; auto].
    + apply NoDup_remove; auto.
Qed.
```

Although not standard library lemmas, the no_nodup_NoDup and Sorted_nodup facts we proved earlier in this file are also both true of nodup_cancel, and proven in almost the same way.

```
Lemma no_nodup_cancel_NoDup : ∀ (A:Type) Aeq_dec (l:list A),
  NoDup l →
  nodup_cancel Aeq_dec l = l.
Proof.
  intros A Aeq_dec l H. induction l.
  - auto.
  - simpl. apply NoDup_cons_iff in H as []. assert (count_occ Aeq_dec l a = 0).
    + apply count_occ_not_In. auto.
    + rewrite H1. simpl. f_equal. rewrite not_In_remove. auto. intro.
      apply nodup_cancel_in in H2. apply H. auto.
Qed.
```

```
Lemma Sorted_nodup_cancel : ∀ (A:Type) Aeq_dec Rel (l:list A),
  Relations_1.Transitive Rel →
```

```
  Sorted Rel l →
  Sorted Rel (nodup_cancel Aeq_dec l).
Proof.
  intros A Aeq_dec Rel l Ht H. apply Sorted_StronglySorted in H; auto.
  apply StronglySorted_Sorted. induction l.
  - auto.
  - simpl. apply StronglySorted_inv in H as []. destruct (even (count_occ Aeq_dec l a)).
    + apply SSorted_cons.
      × apply StronglySorted_remove. apply IHl. apply H.
      × apply Forall_remove. apply Forall_forall. rewrite Forall_forall in H0.
        intros x Hin. apply H0. apply nodup_cancel_in in Hin. auto.
    + apply StronglySorted_remove. apply IHl. apply H.
Qed.
```

An interesting side effect of the "cancelling" behavior of this function is that while the number of occurences of an item may change after calling nodup_cancel, the evenness of the count never will. If an element was odd before there will be one occurence, and if it was even before there will be none.

```
Lemma count_occ_nodup_cancel : ∀ {A Aeq_dec} p (a:A),
  even (count_occ Aeq_dec (nodup_cancel Aeq_dec p) a) =
  even (count_occ Aeq_dec p a).
Proof.
  intros A Aeq_dec p a. induction p as [|b]; auto. simpl.
  destruct (even (count_occ Aeq_dec p b)) eqn:Hb.
  - simpl. destruct (Aeq_dec b a).
    + rewrite e. rewrite count_occ_remove. rewrite e in Hb. repeat rewrite even_succ.
      rewrite ← negb_odd in Hb. rewrite Bool.negb_true_iff in Hb. rewrite Hb. auto.
    + rewrite count_occ_neq_remove; auto.
  - simpl. destruct (Aeq_dec b a).
    + rewrite e. rewrite count_occ_remove. rewrite e in Hb. repeat rewrite even_succ.
      rewrite ← negb_odd in Hb. rewrite Bool.negb_false_iff in Hb. rewrite Hb. auto.
    + rewrite count_occ_neq_remove; auto.
Qed.
```

Permutation_nodup was challenging to prove before, and this version for nodup_cancel faces the same problems. The first and fourth cases are easy, and the second isn't too bad after using count_occ_Permutation. The third case faces the same problems as before, but requires some extra work when transitioning from reasoning about count_occ $(x::l)$ $y$ to count_occ $(y::l)$ $x$.

This is accomplished by using even_succ, negb_odd, and negb_true_iff. In this way, we can convert something saying even $(S\ n)$ = true to even $n$ = false.

```
Lemma nodup_cancel_Permutation : ∀ (A:Type) Aeq_dec (l l':list A),
  Permutation l l' →
```

```
  Permutation (nodup_cancel Aeq_dec l) (nodup_cancel Aeq_dec l').
Proof.
  intros A Aeq_dec l l' H. induction H.
  - auto.
  - simpl. destruct even eqn:Hevn.
    + rewrite (count_occ_Permutation _ _ _ _ _ H) in Hevn. rewrite Hevn.
      apply perm_skip. apply remove_Permutation. apply IHPermutation.
    + rewrite (count_occ_Permutation _ _ _ _ _ H) in Hevn. rewrite Hevn.
      apply remove_Permutation. apply IHPermutation.
  - simpl. destruct (even (count_occ Aeq_dec l x)) eqn:Hevx;
    destruct (even (count_occ Aeq_dec l y)) eqn:Hevy; destruct (Aeq_dec x y).
    + rewrite even_succ. rewrite <- negb_odd in Hevy.
      rewrite Bool.negb_true_iff in Hevy. rewrite Hevy. destruct (Aeq_dec y x);
      try (rewrite e in n; contradiction). rewrite even_succ.
      rewrite <- negb_odd in Hevx. rewrite Bool.negb_true_iff in Hevx.
      rewrite Hevx. simpl. destruct (Aeq_dec y x); try contradiction.
      destruct (Aeq_dec x y); try contradiction. rewrite remove_remove. auto.
    + rewrite Hevy. simpl. destruct (Aeq_dec y x); try (symmetry in e; contradiction).
      destruct (Aeq_dec x y); try contradiction. rewrite Hevx.
      rewrite remove_remove. apply perm_swap.
    + rewrite <- e in Hevy. rewrite Hevy in Hevx. inversion Hevx.
    + rewrite Hevy. simpl. destruct (Aeq_dec y x); try (symmetry in e; contradiction).
      rewrite Hevx. apply perm_skip. rewrite remove_remove. auto.
    + rewrite e in Hevx. rewrite Hevx in Hevy. inversion Hevy.
    + rewrite Hevy. destruct (Aeq_dec y x); try (symmetry in e; contradiction).
      rewrite Hevx. simpl. destruct (Aeq_dec x y); try contradiction.
      apply perm_skip. rewrite remove_remove. auto.
    + rewrite even_succ. rewrite <- negb_odd in Hevy.
      rewrite Bool.negb_false_iff in Hevy. rewrite Hevy. symmetry in e.
      destruct (Aeq_dec y x); try contradiction. rewrite even_succ.
      rewrite <- negb_odd in Hevx. rewrite Bool.negb_false_iff in Hevx.
      rewrite Hevx. rewrite e. auto.
    + rewrite Hevy. destruct (Aeq_dec y x); try (symmetry in e; contradiction).
      rewrite Hevx. rewrite remove_remove. auto.
  - apply Permutation_trans with (l':=(nodup_cancel Aeq_dec l')); auto.
Qed.
```

As mentioned earlier, in the original definition of the function, it was helpful to reverse the order of remove and the recursive call to nodup_cancel. This is possible because these operations are associative, which is proven below.

```
Lemma nodup_cancel_remove_assoc : ∀ {A} Aeq_dec (a:A) p,
  remove Aeq_dec a (nodup_cancel Aeq_dec p) =
  nodup_cancel Aeq_dec (remove Aeq_dec a p).
```

```
Proof.
  intros A Aeq_dec a p. induction p.
  - simpl. auto.
  - simpl. destruct even eqn:Hevn.
    + simpl. destruct (Aeq_dec a a0).
      × rewrite ← e. rewrite not_In_remove; auto. apply remove_In.
      × simpl. rewrite count_occ_neq_remove; auto. rewrite Hevn.
        f_equal. rewrite ← IHp. rewrite remove_remove. auto.
    + destruct (Aeq_dec a a0).
      × rewrite ← e. rewrite not_In_remove; auto. apply remove_In.
      × simpl. rewrite count_occ_neq_remove; auto. rewrite Hevn.
        rewrite remove_remove. rewrite ← IHp. auto.
Qed.
```

The entire point of defining nodup_cancel was so that repeated elements in a list cancel out; clearly then, if an entire list appears twice it will cancel itself out. This proof would be much easier if the order of remove and nodup_cancel was swapped, but the above proof of the two being associative makes it easier to manage.

```
Lemma nodup_cancel_self : ∀ {A} Aeq_dec (l:list A),
  nodup_cancel Aeq_dec (l++l) = [].
Proof.
  intros A Aeq_dec p. induction p.
  - auto.
  - simpl. destruct even eqn:Hevn.
    + rewrite count_occ_app in Hevn. destruct (count_occ Aeq_dec p a) eqn:Hx.
      × simpl in Hevn. destruct (Aeq_dec a a); try contradiction.
        rewrite Hx in Hevn. inversion Hevn.
      × simpl in Hevn. destruct (Aeq_dec a a); try contradiction.
        rewrite Hx in Hevn. rewrite add_comm in Hevn.
        simpl in Hevn. destruct (plus n n) eqn:Help. inversion Hevn.
        replace (plus n n) with (plus 0 (2×n)) in Help.
        pose (even_add_mul_2 0 n). pose (even_succ n0). rewrite ← Help in e1.
        rewrite e0 in e1. simpl in e1. apply even_spec in Hevn. symmetry in e1.
        apply odd_spec in e1. apply (Even_Odd_False _ Hevn) in e1. inversion e1.
        simpl. auto.
    + clear Hevn. rewrite nodup_cancel_remove_assoc. rewrite remove_distr_app.
      simpl. destruct (Aeq_dec a a); try contradiction.
      rewrite ← remove_distr_app. rewrite ← nodup_cancel_remove_assoc.
      rewrite IHp. auto.
Qed.
```

Next up is a useful fact about In that results from nodup_cancel. Because when there's an even number of an element they all get removed, we can say that there will not be any in the resulting list.

```
Lemma not_in_nodup_cancel : ∀ {A Aeq_dec} (m:A) p,
  even (count_occ Aeq_dec p m) = true →
  ¬ In m (nodup_cancel Aeq_dec p).
Proof.
  intros A Aeq_dec m p H. induction p.
  - simpl. auto.
  - intro. simpl in H. destruct (Aeq_dec a m).
    + simpl in H0. rewrite even_succ in H. rewrite ← negb_even in H.
      rewrite Bool.negb_true_iff in H. rewrite ← e in H. rewrite H in H0.
      rewrite e in H0. apply remove_In in H0. inversion H0.
    + apply IHp; auto. simpl in H0. destruct (even (count_occ Aeq_dec p a)).
      × destruct H0; try contradiction. apply In_remove in H0. auto.
      × apply In_remove in H0. auto.
Qed.
```

Similarly to the above lemma, because $a$ will already be removed from $p$ by nodup_cancel, whether or not a remove is added doesn't make a difference.

```
Lemma nodup_extra_remove : ∀ {A Aeq_dec} (a:A) p,
  even (count_occ Aeq_dec p a) = true →
  nodup_cancel Aeq_dec p =
  nodup_cancel Aeq_dec (remove Aeq_dec a p).
Proof.
  intros A Aeq_dec a p H. induction p as [|b]; auto. simpl.
  destruct (Aeq_dec a b).
  - rewrite e in H. simpl in H. destruct (Aeq_dec b b); try contradiction.
    rewrite even_succ in H. rewrite ← negb_even in H.
    rewrite Bool.negb_true_iff in H.
    rewrite H. rewrite nodup_cancel_remove_assoc. rewrite e. auto.
  - simpl. destruct (even (count_occ Aeq_dec p b)) eqn:Hev.
    + rewrite count_occ_neq_remove; auto. rewrite Hev. f_equal.
      rewrite IHp. auto. simpl in H. destruct (Aeq_dec);
      try (symmetry in e; contradiction). auto.
    + rewrite count_occ_neq_remove; auto. rewrite Hev. f_equal.
      apply IHp. simpl in H. destruct (Aeq_dec b a);
      try (symmetry in e; contradiction). auto.
Qed.
```

Lastly, one of the toughest nodup_cancel lemmas. Similarly to nodup_pointless, if nodup_cancel is going to be applied later, there is no need for it to be applied twice. This lemma proves to be very useful when proving that two different polynomials are equal, because, as we will see later, there are often repeated calls to nodup_cancel inside one another. This lemma makes it significantly easier to deal with, as we can remove the redundant nodup_cancels.

This proof proved to be challenging, mostly because it is hard to reason about the parity of the same element in two different lists. In the proof, we begin with induction over $p$, and then move to destructing the count of $a$ in each list. The first case follows easily from the two even hypotheses, count_occ_app, and a couple other lemmas. The second case is almost exactly the same, except $a$ is removed by nodup_cancel and never makes it out front, so the call to perm_skip is removed.

The third case, where $a$ appears and odd number of times in $p$ and an even number of times in $q$, is slightly different, but still solved relatively easily with the use of nodup_extra_remove. The fourth case is by far the hardest. We begin by asserting that, since the count of $a$ in $q$ is odd, there must be at least one, and therefore we can rewrite with In_split to get $q$ into the form of $l1{+}{+}a{+}{+}l2$. We then assert that, since the count of a in $q$ is odd, the count in $l1{+}{+}l2$, or $q$ with one $a$ removed, must surely be even. These facts, combined with remove_distr_app, count_occ_app, and nodup_cancel_remove_assoc, allow us to slowly but surely work $a$ out to the front and eliminate it with perm_skip. All that is left to do at that point is to perform similar steps in the induction hypothesis, so that both *IHp* and our goal are in terms of *l1* and *l2*. *IHp* is then used to finish the proof.

```
Lemma nodup_cancel_pointless : ∀ {A Aeq_dec} (p q:list A),
  Permutation (nodup_cancel Aeq_dec (nodup_cancel Aeq_dec p ++ q))
              (nodup_cancel Aeq_dec (p ++ q)).
Proof.
  intros A Aeq_dec p q. induction p; auto. destruct (even (count_occ Aeq_dec p a))
eqn:Hevp;
  destruct (even (count_occ Aeq_dec q a)) eqn:Hevq.
  - simpl. rewrite Hevp. simpl. rewrite count_occ_app, count_occ_remove. simpl.
    rewrite count_occ_app, even_add, Hevp, Hevq. simpl. apply perm_skip.
    rewrite nodup_cancel_remove_assoc. rewrite remove_pointless.
    rewrite ← nodup_cancel_remove_assoc. apply remove_Permutation. apply IHp.
  - simpl. rewrite Hevp. simpl. rewrite count_occ_app, count_occ_remove. simpl.
    rewrite count_occ_app, even_add, Hevp, Hevq. simpl.
    rewrite nodup_cancel_remove_assoc. rewrite remove_pointless.
    rewrite ← nodup_cancel_remove_assoc. apply remove_Permutation. apply IHp.
  - simpl. rewrite Hevp. rewrite count_occ_app, even_add, Hevp, Hevq. simpl.
    rewrite (nodup_extra_remove a).
    + rewrite remove_pointless. rewrite ← nodup_cancel_remove_assoc.
      apply remove_Permutation. apply IHp.
    + rewrite count_occ_app. rewrite even_add. rewrite count_occ_remove.
      rewrite Hevq. auto.
  - assert (count_occ Aeq_dec q a > 0). destruct (count_occ _ q _).
    inversion Hevq. apply gt_Sn_O. apply count_occ_In in H.
    apply in_split in H as [l1[l2 H]]. rewrite H. simpl nodup_cancel at 2.
    rewrite Hevp. simpl app. rewrite H in IHp. simpl nodup_cancel at 3.
    rewrite count_occ_app. rewrite even_add. rewrite Hevp. rewrite ← H at 2.
    rewrite Hevq. simpl. apply Permutation_trans with (l':=(nodup_cancel
      Aeq_dec (a :: remove Aeq_dec a (nodup_cancel Aeq_dec p) ++ l1 ++ l2))).
```

```
  + apply nodup_cancel_Permutation. rewrite app_assoc. apply Permutation_sym.
    rewrite app_assoc. apply Permutation_middle with (l2:=l2) (l1:=(remove
      Aeq_dec a (nodup_cancel Aeq_dec p) ++ l1)).
  + assert (even (count_occ Aeq_dec (l1++l2) a) = true).
      rewrite H in Hevq. rewrite count_occ_app in Hevq. simpl in Hevq.
      destruct (Aeq_dec a a); try contradiction. rewrite plus_comm in Hevq.
      rewrite plus_Sn_m in Hevq. rewrite even_succ in Hevq.
      rewrite ← negb_even in Hevq. rewrite Bool.negb_false_iff in Hevq.
      rewrite count_occ_app. symmetry. rewrite plus_comm. auto.
    simpl. rewrite count_occ_app. rewrite count_occ_remove. simpl.
    replace (even _) with true. apply perm_skip.
    rewrite (nodup_cancel_remove_assoc _ _ (p++l1++a::l2)).
    repeat rewrite remove_distr_app. simpl; destruct (Aeq_dec a a); try contradiction.
    rewrite nodup_cancel_remove_assoc. rewrite remove_pointless.
    repeat rewrite ← remove_distr_app. repeat rewrite ← nodup_cancel_remove_assoc.
    apply Permutation_trans with (l'':=(nodup_cancel Aeq_dec
    (a :: p ++ l1 ++ l2))) in IHp. apply Permutation_sym in IHp.
    apply Permutation_trans with (l'':=(nodup_cancel Aeq_dec (a :: nodup_cancel
      Aeq_dec p ++ l1 ++ l2))) in IHp.
    simpl in IHp. rewrite count_occ_app, even_add, Hevp in IHp.
    rewrite H0 in IHp. simpl in IHp.
    rewrite count_occ_app, even_add, count_occ_nodup_cancel, Hevp, H0 in IHp.
    simpl in IHp. apply Permutation_sym. apply IHp.
    × apply nodup_cancel_Permutation. rewrite app_assoc. apply Permutation_sym.
      rewrite app_assoc. apply Permutation_middle with
        (l1:=(nodup_cancel Aeq_dec p) ++ l1).
    × apply nodup_cancel_Permutation. rewrite app_assoc. apply Permutation_sym.
      rewrite app_assoc. apply Permutation_middle with (l1:=(p ++ l1)).
Qed.
```

This lemma is simply a reformalization of the above for convenience, which follows simply because of Permutation_app_comm.

```
Lemma nodup_cancel_pointless_r : ∀ {A Aeq_dec} (p q:list A),
  Permutation
    (nodup_cancel Aeq_dec (p ++ nodup_cancel Aeq_dec q))
    (nodup_cancel Aeq_dec (p ++ q)).
Proof.
  intros A Aeq_dec p q. apply Permutation_trans with (l':=(nodup_cancel Aeq_dec (
    nodup_cancel Aeq_dec q ++ p))). apply nodup_cancel_Permutation.
    apply Permutation_app_comm.
  apply Permutation_sym. apply Permutation_trans with (l':=(nodup_cancel
    Aeq_dec (q ++ p))). apply nodup_cancel_Permutation.
    apply Permutation_app_comm. apply Permutation_sym.
```

```
  apply nodup_cancel_pointless.
Qed.
```

An interesting side effect of nodup_cancel_pointless is that now we can show that nodup_cancel almost "distributes" over app. More formally, to prove that the nodup_cancel of two lists appended together is a permutation of nodup_cancel applied to two other lists appended, it is sufficient to show that the first of each and the second of each are permutations after applying nodup_cancel to them individually.

```
Lemma nodup_cancel_app_Permutation : ∀ {A Aeq_dec} (a b c d:list A),
  Permutation (nodup_cancel Aeq_dec a) (nodup_cancel Aeq_dec b) →
  Permutation (nodup_cancel Aeq_dec c) (nodup_cancel Aeq_dec d) →
  Permutation (nodup_cancel Aeq_dec (a ++ c)) (nodup_cancel Aeq_dec (b ++ d)).
Proof.
  intros A Aeq_dec a b c d H H0. rewrite ← (nodup_cancel_pointless a),
  ← (nodup_cancel_pointless b), ← (nodup_cancel_pointless_r _ c),
  ← (nodup_cancel_pointless_r _ d). apply nodup_cancel_Permutation.
  apply Permutation_app; auto.
Qed.
```

### 5.4.3 Comparing Parity of Lists: parity_match

The final major definition over lists we wrote is parity_match. parity_match is closely related to nodup_cancel, and allows us to make statements about lists being equal after applying nodup_cancel to them. Clearly, if an element appears an even number of times in both lists, then it won't appear at all after nodup_cancel, and if an element appears an odd number of times in both lists, then it will appear once after nodup_cancel. The ultimate goal of creating this definition is to prove a lemma that if the parity of two lists matches, they are permutations of each other after applying nodup_cancel.

The definition simply states that for all elements, the parity of the number of occurences in each list is equal.

```
Definition parity_match {A} Aeq_dec (l m:list A) : Prop :=
  ∀ x, even (count_occ Aeq_dec l x) = even (count_occ Aeq_dec m x).
```

A useful lemma in working towards this proof is that if the count of every variable in a list is even, then there will be no variables in the resulting list. This is relatively easy to prove, as we have already proven not_in_nodup_cancel and can contradict away the other cases.

```
Lemma even_nodup_cancel : ∀ {A Aeq_dec} (p:list A),
  (∀ x, even (count_occ Aeq_dec p x) = true) →
  (∀ x, ¬ In x (nodup_cancel Aeq_dec p)).
Proof.
  intros A Aeq_dec p H m. intro. induction p.
  - inversion H0.
```

```
- simpl in *. pose (H m) as H1. symmetry in H1. destruct (Aeq_dec a m).
   + symmetry in H1. rewrite ← e in H1. rewrite even_succ in H1. rewrite ←
negb_even in H1.
      rewrite Bool.negb_true_iff in H1. rewrite H1 in H0. rewrite e in H0.
      apply remove_In in H0. inversion H0.
   + destruct (even (count_occ Aeq_dec p a)).
     × destruct H0; try contradiction. apply In_remove in H0. symmetry in H1.
       apply not_in_nodup_cancel in H1. contradiction.
     × apply In_remove in H0. symmetry in H1. apply not_in_nodup_cancel in H1.
       contradiction.
Qed.
```

The above lemma can then be used in combination with nothing_in_empty to easily prove parity_match_empty, which will be useful in two cases of our goal lemma.

```
Lemma parity_match_empty : ∀ {A Aeq_dec} (q:list A),
  parity_match Aeq_dec [] q →
  Permutation [] (nodup_cancel Aeq_dec q).
Proof.
  intros A Aeq_dec q H. unfold parity_match in H. simpl in H.
  symmetry in H. pose (even_nodup_cancel q H). apply nothing_in_empty in n.
  rewrite n. auto.
Qed.
```

The parity_match definition is also reflexive, symmetric, and transitive, and knowing this will make future proofs easier.

```
Lemma parity_match_refl : ∀ {A Aeq_dec} (l:list A),
  parity_match Aeq_dec l l.
Proof.
  intros A Aeq_dec l. unfold parity_match. auto.
Qed.

Lemma parity_match_sym : ∀ {A Aeq_dec} (l m:list A),
  parity_match Aeq_dec l m ↔ parity_match Aeq_dec m l.
Proof.
  intros l m. unfold parity_match. split; intros H x; auto.
Qed.

Lemma parity_match_trans : ∀ {A Aeq_dec} (p q r:list A),
  parity_match Aeq_dec p q →
  parity_match Aeq_dec q r →
  parity_match Aeq_dec p r.
Proof.
  intros A Aeq_dec p q r H H0. unfold parity_match in *. intros x.
  rewrite H. rewrite H0. auto.
Qed.
```

```
Hint Resolve parity_match_refl parity_match_sym parity_match_trans.
```

There are also a few interesting facts that can be proved about elements being consed onto lists in a parity_match. First is that if the parity of two lists is equal, then the parities will also be equal after adding another element to the front, and vice versa.

```
Lemma parity_match_cons : ∀ {A Aeq_dec} (a:A) l1 l2,
  parity_match Aeq_dec (a::l1) (a::l2) ↔
  parity_match Aeq_dec l1 l2.
Proof.
  intros A Aeq_dec a l1 l2. unfold parity_match. split; intros H x.
  - pose (H x). symmetry in e. simpl in e. destruct (Aeq_dec a x); auto.
    repeat rewrite even_succ in e. repeat rewrite ← negb_even in e.
    apply Bool.negb_sym in e. rewrite Bool.negb_involutive in e. auto.
  - simpl. destruct (Aeq_dec a x); auto.
    repeat rewrite even_succ. repeat rewrite ← negb_even.
    apply Bool.negb_sym. rewrite Bool.negb_involutive. auto.
Qed.
```

Similarly, adding the same element twice to a list does not change the parities of any elements in the list.

```
Lemma parity_match_double : ∀ {A Aeq_dec} (a:A) l,
  parity_match Aeq_dec (a::a::l) l.
Proof.
  intros A Aeq_dec a l. unfold parity_match. intros x. simpl.
  destruct (Aeq_dec a x).
  - rewrite even_succ. rewrite odd_succ. auto.
  - auto.
Qed.
```

The last cons parity_match lemma states that if you remove an element from one list and add it to the other, the parity will not be affected. This follows because if they both had an even number of $a$ before they will both have an odd number after, and if it was odd before it will be even after.

```
Lemma parity_match_cons_swap : ∀ {A Aeq_dec} (a:A) l1 l2,
  parity_match Aeq_dec (a::l1) l2 →
  parity_match Aeq_dec l1 (a::l2).
Proof.
  intros A Aeq_dec a l1 l2 H. apply (parity_match_cons a) in H.
  apply parity_match_sym in H. apply parity_match_trans with (r:=l1) in H.
  apply parity_match_sym in H. auto. apply parity_match_double.
Qed.
```

This next lemma states that if we know that some element $a$ appears in the *rest* of the list an even number of times, than clearly it appears in $l2$ an odd number of times and must be in the list.

```
Lemma parity_match_In : ∀ {A Aeq_dec} (a:A) l1 l2,
  even (count_occ Aeq_dec l1 a) = true →
  parity_match Aeq_dec (a::l1) l2 →
  In a l2.
Proof.
  intros A Aeq_dec a l1 l2 H H0. apply parity_match_cons_swap in H0.
  rewrite H0 in H. simpl in H. destruct (Aeq_dec a a); try contradiction.
  rewrite even_succ in H. rewrite ← negb_even in H. rewrite Bool.negb_true_iff in H.
  assert (count_occ Aeq_dec l2 a > 0). destruct count_occ. inversion H.
  apply gt_Sn_O. apply count_occ_In in H1. auto.
Qed.
```

The last fact to prove before attempting the big lemma is that if two lists are permutations of each other, then their parities must match because they contain the same elements the same number of times.

```
Lemma Permutation_parity_match : ∀ {A Aeq_dec} (p q:list A),
  Permutation p q → parity_match Aeq_dec p q.
Proof.
  intros A Aeq_dec p q H. induction H.
  - auto.
  - apply parity_match_cons. auto.
  - repeat apply parity_match_cons_swap. unfold parity_match. intros x0.
    simpl. destruct Aeq_dec; destruct Aeq_dec;
    repeat (rewrite even_succ; rewrite odd_succ); auto.
  - apply parity_match_trans with (q:=l'); auto.
Qed.
```

Finally, the big one. The first three cases are straightforward, especially now that we have already proven parity_match_empty. The third case is more complicated. We begin by destructing if $a$ and $a0$ are equal. In the case that they are, the proof is relatively straightforward; parity_match_cons, perm_skip, and remove_Permutation take care of it.

In the case that they are not equal, we next destruct if the number of occurences is even or not. If it is odd, we can use parity_match_In and In_split to rewrite l2 in terms of $a$. From there, we use permutation facts to rearrange $a$ to be at the front, and the rest of the proof is similar to the proof when $a$ and $a0$ are equal.

The final case is when they are not equal and the number of occurences is even. After using parity_match_cons_swap, we can get to a point where we know that $a$ appears in $q{+}{+}a0$ an even number of times. This means that $a$ will not be in $q{+}{+}a0$ after applying nodup_cancel, so we can rewrite with not_In_remove in the reverse direction to get the two sides of the permutation goal to be more similar. Then, because it is wrapped in remove $a$, we can clearly add an $a$ on the inside without it having any effect. Then all that is left is to apply remove_Permutation, and we end up with a goal matching the induction hypothesis.

This lemma is very powerful, especially when dealing with nodup_cancel with functions applied to the elements of a list. This will come into play later in this file.

```
Lemma parity_nodup_cancel_Permutation : ∀ {A Aeq_dec} (p q:list A),
  parity_match Aeq_dec p q →
  Permutation (nodup_cancel Aeq_dec p) (nodup_cancel Aeq_dec q).
Proof.
  intros A Aeq_dec p q H. generalize dependent q. induction p; induction q; intros.
  - auto.
  - simpl nodup_cancel at 1. apply parity_match_empty. auto.
  - simpl nodup_cancel at 2. apply Permutation_sym. apply parity_match_empty.
    apply parity_match_sym. auto.
  - clear IHq. destruct (Aeq_dec a a0).
    + rewrite e. simpl. rewrite e in H. apply parity_match_cons in H.
      destruct even eqn:Hev; rewrite H in Hev; rewrite Hev.
      × apply perm_skip. apply remove_Permutation. auto.
      × apply remove_Permutation. auto.
    + simpl nodup_cancel at 1. destruct even eqn:Hev.
      × assert (Hev':=Hev). apply parity_match_In with (l2:=(a0::q)) in Hev; auto.
        destruct Hev. symmetry in H0. contradiction. apply In_split in H0 as [l1[l2
H0]].
        rewrite H0. apply Permutation_sym. apply Permutation_trans with (l':=(
          nodup_cancel Aeq_dec (a::l2++a0::l1))). apply nodup_cancel_Permutation.
          rewrite app_comm_cons. apply (Permutation_app_comm).
        simpl. rewrite H0 in H. apply parity_match_trans with (r:=(a::l2++a0::l1))
in H.
        apply parity_match_cons in H. rewrite H in Hev'. rewrite Hev'.
        apply perm_skip. apply remove_Permutation. apply Permutation_sym.
        apply IHp. auto. rewrite app_comm_cons. apply Permutation_parity_match.
        apply Permutation_app_comm.
      × apply parity_match_cons_swap in H. rewrite H in Hev. assert (Hev2:=Hev).
        rewrite count_occ_Permutation with (l':=(a::q++[a0])) in Hev. simpl in Hev.
        destruct (Aeq_dec a a); try contradiction. rewrite even_succ in Hev.
        rewrite ← negb_even in Hev. rewrite Bool.negb_false_iff in Hev.
        rewrite ← (not_In_remove _ Aeq_dec a).
        assert (∀ l, remove Aeq_dec a (nodup_cancel Aeq_dec (l)) =
          remove Aeq_dec a (nodup_cancel Aeq_dec (a::l))).
          intros l. simpl. destruct (even (count_occ _ l a)).
          simpl. destruct (Aeq_dec a a); try contradiction.
          rewrite (not_In_remove _ _ _(remove _ _ _)). auto. apply remove_In.
          rewrite (not_In_remove _ _ _(remove _ _ _)). auto. apply remove_In.
        rewrite (H0 (a0::q)). apply remove_Permutation. apply IHp. auto.
        apply not_in_nodup_cancel. rewrite count_occ_Permutation with (l':=(a0::q))
in Hev.
        auto. replace (a0::q) with ([a0]++q); auto. apply Permutation_app_comm.
```

```
    apply perm_skip. replace (a0 :: q) with ([a0]++q); auto. apply Permutation_app_comm.
Qed.
```

## 5.5 Combining nodup_cancel and Other Functions

### 5.5.1 Using nodup_cancel over map

Our next goal is to prove things about the relation between nodup_cancel and map over lists. In particular, we want to prove a lemma similar to nodup_cancel_pointless, that allows us to remove redundant nodup_cancels.

The challenging part of proving this lemma is that it is often hard to reason about how, for example, the number of times $a$ appears in $p$ relates to the number of times $f\ a$ appears in map $f\ p$. Many of the functions we map across lists in practice are not one-to-one, meaning that there could be some $b$ such that $f\ a = f\ b$. However, at the end of the day, these repeated elements will cancel out with each other and the parities will match, hence why parity_nodup_cancel_Permutation is extremely useful.

To begin, we need to prove a couple facts comparing the number of occurences of elements in a list. The first lemma states that the number of times some $a$ appears in $p$ is less than or equal to the number of times $f\ a$ appears in map $f\ p$.

```
Lemma count_occ_map_lt : ∀ {A Aeq_dec} p (a:A) f,
  count_occ Aeq_dec p a ≤ count_occ Aeq_dec (map f p) (f a).
Proof.
  intros A Aeq_dec p a f. induction p. auto. simpl. destruct Aeq_dec.
  - rewrite e. destruct Aeq_dec; try contradiction. simpl. apply le_n_S. auto.
  - destruct Aeq_dec; auto.
Qed.
```

Building off this idea, the next lemma states that the number of times $f\ a$ appears in map $f\ p$ with $a$ removed is equal to the count of $f\ a$ in map $f\ p$ minus the count of $a$ in $p$.

```
Lemma count_occ_map_sub : ∀ {A Aeq_dec} f (a:A) p,
  count_occ Aeq_dec (map f (remove Aeq_dec a p)) (f a) =
  count_occ Aeq_dec (map f p) (f a) - count_occ Aeq_dec p a.
Proof.
  intros A Aeq_dec f a p. induction p; auto. simpl. destruct Aeq_dec.
  - rewrite e. destruct Aeq_dec; try contradiction. destruct Aeq_dec;
    try contradiction. simpl. rewrite ← e. auto.
  - simpl. destruct Aeq_dec.
    + destruct Aeq_dec. symmetry in e0; contradiction. rewrite IHp.
      rewrite sub_succ_l. auto. apply count_occ_map_lt.
    + destruct Aeq_dec. symmetry in e; contradiction. auto.
Qed.
```

It is also true that if there is some $x$ that is *not* equal to $f\ a$, then the count of that $x$ in map $f\ p$ is the same as the count of $x$ in map $f\ p$ with $a$ removed.

```
Lemma count_occ_map_neq_remove : ∀ {A Aeq_dec} f (a:A) p x,
  x ≠ (f a) →
  count_occ Aeq_dec (map f (remove Aeq_dec a p)) x =
  count_occ Aeq_dec (map f p) x.
Proof.
  intros. induction p as [|b]; auto. simpl. destruct (Aeq_dec a b).
  - destruct Aeq_dec. rewrite ← e in e0. symmetry in e0. contradiction.
    auto.
  - simpl. destruct Aeq_dec; auto.
Qed.
```

The next lemma is similar to count_occ_map_lt, except it involves some $b$ where $a$ is not equal to $b$, but $f\ a = f\ b$. Then clearly, the sum of $a$ in $p$ and $b$ in $p$ is less than the count of $f\ a$ in map $f\ p$.

```
Lemma f_equal_sum_lt : ∀ {A Aeq_dec} f (a:A) b p,
  b ≠ a → (f a) = (f b) →
  count_occ Aeq_dec p b +
  count_occ Aeq_dec p a ≤
  count_occ Aeq_dec (map f p) (f a).
Proof.
  intros A Aeq_dec f a b p Hne Hfe. induction p as [|c]; auto. simpl. destruct Aeq_dec.
  - rewrite e. destruct Aeq_dec; try contradiction. rewrite Hfe.
    destruct Aeq_dec; try contradiction. simpl. apply le_n_S.
    rewrite ← Hfe. auto.
  - destruct Aeq_dec.
    + rewrite e. destruct Aeq_dec; try contradiction. rewrite plus_comm.
      simpl. rewrite plus_comm. apply le_n_S. auto.
    + destruct Aeq_dec.
      × apply le_S. auto.
      × auto.
Qed.
```

For the next lemma, we once again try to compare the count of $a$ to the count of $f\ a$, but also involve nodup_cancel. Clearly, there is no way for there to be more $a$'s in $p$ than $f\ a$'s in map $f\ p$ even with the addition of nodup_cancel.

```
Lemma count_occ_nodup_map_lt : ∀ {A Aeq_dec} p f (a:A),
  count_occ Aeq_dec (nodup_cancel Aeq_dec p) a ≤
  count_occ Aeq_dec (map f (nodup_cancel Aeq_dec p)) (f a).
Proof.
  intros A Aeq_dec p f a. induction p as [|b]; auto. simpl. destruct even eqn:Hev.
  - simpl. destruct Aeq_dec.
```

```
    + rewrite e. destruct Aeq_dec; try contradiction. apply le_n_S. auto.
      rewrite count_occ_remove. apply le_0_l.
    + rewrite count_occ_neq_remove; auto. rewrite not_In_remove.
      destruct Aeq_dec; firstorder. apply not_in_nodup_cancel; auto.
  - destruct (Aeq_dec b a) eqn:Hba.
    + rewrite e. rewrite count_occ_remove. apply le_0_l.
    + rewrite count_occ_neq_remove; auto. destruct (Aeq_dec (f b) (f a)) eqn:Hfba.
      × rewrite ← e. rewrite count_occ_map_sub. rewrite e. apply le_add_le_sub_l.
        apply f_equal_sum_lt; auto.
      × rewrite count_occ_map_neq_remove; auto.
Qed.
```

All of these lemmas now come together for the core one, a variation of nodup_cancel_pointless but involving map $f$. We begin by applying parity_nodup_cancel_Permutation, and destructing if $a$ appears in $p$ an even number of times or not.

The even case is relatively easy to prove, and only involves using the usual combination of even_succ, not_In_remove, and not_in_nodup_cancel.

The odd case is trickier, and where we involve all of the newly proved lemmas. If $x$ and $f\ a$ are not equal, the proof follows just from count_occ_map_neq_remove and the induction hypothesis.

If they are equal, we begin by rewriting with count_occ_map_sub and even_sub. After a few more rewrites, it becomes the case that we need to prove that the boolean equivalence of the parities of $f\ a$ in map $f\ p$ and $a$ in $p$ is equal to the negated parity of $f\ a$ in map $f\ p$. Because we know that $a$ appears in $p$ an odd number of times from destructing even earlier, this follows immediately.

```
Lemma nodup_cancel_map : ∀ {A Aeq_dec} (p:list A) f,
  Permutation
    (nodup_cancel Aeq_dec (map f (nodup_cancel Aeq_dec p)))
    (nodup_cancel Aeq_dec (map f p)).
Proof.
  intros A Aeq_dec p f. apply parity_nodup_cancel_Permutation. unfold parity_match.
  intros x. induction p; auto. simpl. destruct (even (count_occ _ p a)) eqn:Hev.
  - simpl. destruct Aeq_dec.
    + repeat rewrite even_succ. repeat rewrite ← negb_even. rewrite not_In_remove.
      rewrite IHp. auto. apply not_in_nodup_cancel. auto.
    + rewrite not_In_remove. apply IHp. apply not_in_nodup_cancel. auto.
  - simpl. destruct Aeq_dec.
    + rewrite ← e. rewrite count_occ_map_sub. rewrite even_sub. rewrite ← e in
IHp.
      rewrite IHp. rewrite count_occ_nodup_cancel. rewrite Hev. rewrite even_succ.
      rewrite ← negb_even. destruct (even (count_occ _ (map f p) _)); auto.
      apply count_occ_nodup_map_lt.
    + rewrite count_occ_map_neq_remove; auto.
```

Qed.

### 5.5.2 Using nodup_cancel over concat map

Similarly to map, the same property of not needing repeated nodup_cancels applies when the lists are being concatenated and mapped over. This final section of the file seeks to, in very much the same way as earlier, prove this.

We begin with a simple lemma about math that will come into play soon - if a number is less than or equal to 1, then it is either 0 or 1. This is immediately solved with firstorder logic.

```
Lemma n_le_1 : ∀ n,
  n ≤ 1 → n = 0 ∨ n = 1.
Proof.
  intros n H. induction n; firstorder.
Qed.
```

The main difference between this section and the section about map is that all of the functions being mapped will clearly be returning lists as their output, and then being concatenated with the rest of the result. This makes things slightly harder, as we can't reason about the number of times, for example, some $f\ a$ appears in a list. Instead, we have to reason about the number of times that some $x$ appears in a list, where $x$ is one of the elements of the list $f\ a$.

In practice, these lemmas are only going to be applied in situations where every $f\ a$ has no duplicates in it. In other words, as the lemma above states, there will be either 0 or 1 of each x in a list. The next two lemmas prove some consequences of this.

First is that if the count of $x$ in $f\ a$ is 0, then clearly removing $a$ from some list $p$ will not affect the count of $x$ in the concatenated version of the list.

```
Lemma count_occ_map_sub_not_in : ∀ {A Aeq_dec} f (a:A) p,
  ∀ x, count_occ Aeq_dec (f a) x = 0 →
  count_occ Aeq_dec (concat (map f (remove Aeq_dec a p))) x =
  count_occ Aeq_dec (concat (map f p)) x.
Proof.
  intros A Aeq_dec f a p x H. induction p as [|b]; auto. simpl.
  rewrite count_occ_app. destruct Aeq_dec.
  - rewrite e in H. rewrite H. firstorder.
  - simpl. rewrite count_occ_app. auto.
Qed.
```

On the other hand, if the count of some $x$ in $f\ a$ is 1, then the count of $a$ in the original list must be less than or equal to the count of $x$ in the final list, depending on if some $b$ exists such that $f\ a$ also contains $x$. More useful is the fact that if $x$ appears once in $f\ x$, the count of $x$ in the final list with $a$ removed is equal to the count of $x$ in the final list minus the count of $a$ in the list. Both of these proofs are relatively straightforward, and mostly follow from firstorder logic.

```
Lemma count_occ_concat_map_lt : ∀ {A Aeq_dec} p (a:A) f x,
  count_occ Aeq_dec (f a) x = 1 →
  count_occ Aeq_dec p a ≤ count_occ Aeq_dec (concat (map f p)) x.
Proof.
  intros A Aeq_dec p a f x H. induction p. auto. simpl. destruct Aeq_dec.
  - rewrite e. rewrite count_occ_app. rewrite H. simpl. firstorder.
  - rewrite count_occ_app. induction (count_occ Aeq_dec (f a0) x); firstorder.
Qed.

Lemma count_occ_map_sub_in : ∀ {A Aeq_dec} f (a:A) p,
  ∀ x, count_occ Aeq_dec (f a) x = 1 →
  count_occ Aeq_dec (concat (map f (remove Aeq_dec a p))) x =
  count_occ Aeq_dec (concat (map f p)) x - count_occ Aeq_dec p a.
Proof.
  intros A Aeq_dec f a p x H. induction p as [|b]; auto. simpl. destruct Aeq_dec.
  - rewrite e. destruct Aeq_dec; try contradiction. rewrite count_occ_app.
    rewrite e in H. rewrite H. simpl. rewrite ← e. auto.
  - simpl. destruct Aeq_dec. symmetry in e. contradiction.
    repeat rewrite count_occ_app. rewrite IHp. rewrite add_sub_assoc. auto.
    apply count_occ_concat_map_lt; auto.
Qed.
```

Continuing the pattern of proving similar facts as we did during the map proof, we now prove a version of f_equal_sum_lt involving concat. This lemma states that, if we know there will be no duplicates in $f\ x$ for all $x$, and that there are some $a$ and $b$ such that they are not equal but $x$ in in both $f\ a$ and $f\ b$, then clearly the sum of the count of $a$ and the count of $b$ is less than or equal to the count of $x$ in the list after applying the function and concatenating.

```
Lemma f_equal_concat_sum_lt : ∀ {A Aeq_dec} f (a:A) b p x,
  b ≠ a →
  (∀ x, NoDup (f x)) →
  count_occ Aeq_dec (f a) x = 1 →
  count_occ Aeq_dec (f b) x = 1 →
  count_occ Aeq_dec p b +
  count_occ Aeq_dec p a ≤
  count_occ Aeq_dec (concat (map f p)) x.
Proof.
  intros A Aeq_dec f a b p x Hne Hnd Hfa Hfb. induction p as [|c]; auto. simpl.
  destruct Aeq_dec.
  - rewrite e. destruct Aeq_dec; try contradiction. rewrite count_occ_app.
    firstorder.
  - destruct Aeq_dec.
    + rewrite e. rewrite count_occ_app. firstorder.
    + rewrite count_occ_app. pose (Hnd c). rewrite (NoDup_count_occ Aeq_dec) in n1.
```

```
    pose (n1 x). apply n_le_1 in l. clear n1. destruct l; firstorder.
Qed.
```

The last step before we are able to prove nodup_cancel_concat_map is to actually involve nodup_cancel rather than just remove. This lemma states that given $f\ x$ has no duplicates and $a$ appears once in $f\ a$, the count of $a$ in $p$ after applying nodup_cancel is less than or equal to the count of $x$ after applying concat map and nodup_cancel.

The first cases, when the count is even, are relatively straightforward. The second cases, when the count is odd, are slightly more complicated. We destruct if $a$ and $b$ (where $b$ is our induction element) are equal. If they are, then the proof is solved by firstorder logic. On the other hand, if they are not, we make use of our n_le_1 fact proved before to find out how many times $x$ appears in $f\ b$. If it is zero, then we rewrite with the 0 fact proved earlier and are done. In the final case, we rewrite with the 1 subtraction fact we proved earlier, and it follows from f_equal_concat_sum_lt.

```
Lemma count_occ_nodup_concat_map_lt : ∀ {A Aeq_dec} p f (a:A) x,
  (∀ x, NoDup (f x)) →
  count_occ Aeq_dec (f a) x = 1 →
  count_occ Aeq_dec (nodup_cancel Aeq_dec p) a ≤
  count_occ Aeq_dec (concat (map f (nodup_cancel Aeq_dec p))) x.
Proof.
  intros A Aeq_dec p f a x Hn H. induction p as [|b]; auto. simpl. destruct even
eqn:Hev.
  - simpl. destruct Aeq_dec.
    + rewrite e. rewrite count_occ_remove, count_occ_app. rewrite H. firstorder.
    + rewrite count_occ_neq_remove; auto. rewrite not_In_remove.
      rewrite count_occ_app. firstorder. apply not_in_nodup_cancel. auto.
  - destruct (Aeq_dec b a) eqn:Hba.
    + rewrite e. rewrite count_occ_remove. firstorder.
    + rewrite count_occ_neq_remove; auto. assert (Hn1:=(Hn b)).
      rewrite (NoDup_count_occ Aeq_dec) in Hn1. assert (Hn2:=(Hn1 x)).
      clear Hn1. apply n_le_1 in Hn2. destruct Hn2.
      × rewrite count_occ_map_sub_not_in; auto.
      × apply (count_occ_map_sub_in _ _ (nodup_cancel Aeq_dec p)) in H0 as H1.
        rewrite H1. apply le_add_le_sub_l. apply f_equal_concat_sum_lt; auto.
Qed.
```

Finally, the proof we've been building up to. Once again, we begin the proof by converting to a parity_match problem and then perform induction on the list. The case where $a$ appears an even number of times in the list is easy, and follows from the same combination of count_occ_app and even_add that we have used before.

The case where $a$ appears an odd number of times is slightly more complex. Once again, we apply n_le_1 to determine how many times our $x$ appears in $f\ a$. If it is zero times, we use count_occ_map_sub_not_in like above, and then the induction hypothesis solves it. If $x$ appears once in $f\ a$, we instead use count_occ_map_sub_in combined with even_sub. Then, after rewriting with the induction hypothesis, we can easily solve the lemma with the use of count_occ_nodup_cancel.

```
Lemma nodup_cancel_concat_map : ∀ {A Aeq_dec} (p:list A) f,
  (∀ x, NoDup (f x)) →
  Permutation
    (nodup_cancel Aeq_dec (concat (map f (nodup_cancel Aeq_dec p))))
    (nodup_cancel Aeq_dec (concat (map f p))).
Proof.
  intros A Aeq_dec p f H. apply parity_nodup_cancel_Permutation. unfold parity_match.
  intros x. induction p; auto. simpl. destruct (even (count_occ _ p a)) eqn:Hev.
  - simpl. repeat rewrite count_occ_app. repeat rewrite even_add. rewrite not_In_remove.
    rewrite IHp. auto. apply not_in_nodup_cancel. auto.
  - assert (H0:=(H a)). rewrite (NoDup_count_occ Aeq_dec) in H0.
    assert (H1:=(H0 x)). clear H0. apply n_le_1 in H1. rewrite count_occ_app.
    rewrite even_add. destruct H1.
    + apply (count_occ_map_sub_not_in _ _ (nodup_cancel Aeq_dec p)) in H0 as H1.
      rewrite H0, H1, IHp. simpl.
      destruct (even (count_occ _ (concat (map f p)) x)); auto.
    + apply (count_occ_map_sub_in _ _ (nodup_cancel Aeq_dec p)) in H0 as H1.
      rewrite H0, H1, even_sub, IHp. simpl. rewrite count_occ_nodup_cancel. rewrite
Hev.
      destruct (even (count_occ _ (concat (map f p)) x)); auto.
      apply count_occ_nodup_concat_map_lt; auto.
Qed.
```

# Chapter 6

# Library B_Unification.poly

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

Require Export list_util.
Require Export terms.
```

## 6.1 Monomials and Polynomials

### 6.1.1 Data Type Definitions

Now that we have defined those functions over lists and proven all of those facts about them, we can begin to apply all of them to our specific project of unification. The first step is to define the data structures we plan on using.

As mentioned earlier, because of the ten axioms that hold true during B-Unification, we can represent all possible terms with lists of lists of numbers. The numbers represent variables, and a list of variables is a monomial, where each variable is multiplied together. A polynomial, then, is a list of monomials where each monomial is added together.

In this representation, the term 0 is represented as the empty polynomial, and the term 1 is represented as the polynomial containing only the empty monomial.

In addition to the definitions of mono and poly, we also have a definition for mono_eq_dec; this is a proof of decidability of monomials. This makes use of a special Coq data structure that allows this to be used as a comparison function - for example, we can `destruct` (mono_eq_dec *a* *b*) to compare the two cases where a = b and a <> b. In addition to being useful in some proofs, this is also needed by some functions, such as remove and count_occ, since they compare monomials.

```
Definition mono := list var.
Definition mono_eq_dec := (list_eq_dec Nat.eq_dec).

Definition poly := list mono.
```

### 6.1.2 Comparisons of monomials and polynomials

In order to easily compare monomials, we make use of the lex function we defined at the beginning of the list_util file. For convenience, we also define mono_lt, which is a proposition that states that some monomial is less than another.

```
Definition mono_cmp := lex compare.

Definition mono_lt m n := mono_cmp m n = Lt.
```

A simple but useful definition is vars, which allows us to take any polynomial and get a list of all the variables in it. This is simply done by concatenating all of the monomials into one large list of variables and removing any repeated variables.

Clearly then, there will never be any duplicates in the vars of some polynomial.

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

Lemma NoDup_vars : ∀ (p : poly),
  NoDup (vars p).
Proof.
  intros p. unfold vars. apply NoDup_nodup.
Qed.
```

This next lemma allows us to convert from a statement about vars to a statement about the monomials themselves. If some variable $x$ is not in the variables of a polynomial $p$, then every monomial in $p$ must not contain $x$.

```
Lemma in_mono_in_vars : ∀ x p,
  (∀ m : mono, In m p → ¬ In x m) ↔ ¬ In x (vars p).
Proof.
  intros x p. split.
  - intros H. induction p.
    + simpl. auto.
    + unfold not in *. intro. apply IHp.
      × intros m Hin. apply H. intuition.
      × unfold vars in *. apply nodup_In in H0. apply nodup_In. simpl in H0.
        apply in_app_or in H0. destruct H0.
        – exfalso. apply (H a). intuition. auto.
        – auto.
  - intros H m Hin Hin'. apply H. clear H. induction p.
    + inversion Hin.
```

```
    + unfold vars in *. rewrite nodup_In. rewrite nodup_In in IHp. simpl.
      apply in_or_app. destruct Hin.
      × left. rewrite H. auto.
      × auto.
Qed.
```

### 6.1.3 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 lists of natural numbers that, for ease of comparison, are sorted least to greatest. A small sublety is that we are insisting they are sorted with lt, meaning less than, rather than *le*, or less than or equal to. This way, the Sorted predicate will insist that each number is *less than* the one following it, thereby preventing any values from being equal to each other. In this way, we simultaneously enforce the sorting and lack of duplicated values in a monomial.

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

Polynomials are sorted lists of lists, where all of the lists in the polynomial are monomials. Similarly to the last example, we use mono_lt to simultaneously enforce sorting and no-duplicates.

```
Definition is_poly (p : poly) : Prop :=
  Sorted mono_lt p ∧ ∀ m, In m p → is_mono m.

Hint Unfold is_mono is_poly.
Hint Resolve NoDup_cons NoDup_nil Sorted_cons.
```

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

```
Lemma mono_order : ∀ x y m,
  is_mono (x :: y :: m) →
  x < y.
Proof.
  unfold is_mono.
  intros x y m H.
  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 x m H. 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, and if m :: p is a polynomial, we know both that p is a polynomial and that m is a monomial.

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

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, the polynomial representation for one as we described before is a polynomial, and a singleton variable is a polynomial.

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

```
Qed.

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

Lemma one_is_poly :
  is_poly [[]].
Proof.
  unfold is_poly. split.
  - auto.
  - intro. intro. simpl in H. destruct H.
    + rewrite ← H. apply nil_is_mono.
    + inversion H.
Qed.

Lemma var_is_poly : ∀ x,
  is_poly [[x]].
Proof.
  intros x. unfold is_poly. split.
  - apply Sorted_cons; auto.
  - intros m H. simpl in H; destruct H; inversion H.
    unfold is_mono. auto.
Qed.
```

In unification, a common concept is a *ground term*, or a term that contains no variables. If some polynomial is a ground term, then it must either be equal to 0 or 1.

```
Lemma no_vars_is_ground : ∀ p,
  is_poly p →
  vars p = [] →
  p = [] ∨ p = [[]].
Proof.
  intros p H H0. induction p.
  - auto.
  - induction a.
    + destruct IHp.
      × apply poly_cons in H. apply H.
      × unfold vars in H0. simpl in H0. apply H0.
      × rewrite H1. auto.
      × rewrite H1 in H. unfold is_poly in H. destruct H. inversion H.
        inversion H6. inversion H8.
```

```
    + unfold vars in H0. simpl in H0. destruct in_dec in H0.
      × rewrite ← nodup_In in i. rewrite H0 in i. inversion i.
      × inversion H0.
Qed.
Hint Resolve mono_order mono_cons poly_order poly_cons nil_is_mono nil_is_poly
  var_is_poly one_is_poly.
```

## 6.2 Sorted Lists and Sorting

Clearly, because we want to maintain that our monomials and polynomials are sorted at all times, we will be dealing with Coq's Sorted proposition a lot. In addition, not every list we want to operate on will already be perfectly sorted, so it is often necessary to sort lists ourselves. This next section serves to give us all of the tools necessary to operate on sorted lists.

### 6.2.1 Sorting Lists

In order to sort our lists, we will make use of the Sorting module in the standard library, which implements a version of merge sort.

For sorting variables in a monomial, we can simply reuse the already provided *NatSort* module.

```
Module Import VarSort := NatSort.
```

Sorting the monomials in a polynomial is slightly more complicated, but still straightforward thanks to the Sorting module. First, we need to define a MonoOrder, which must be a total less-than-or-equal-to comparator.

This is accomplished by using our mono_cmp defined earlier, and simply returning true for either less than or equal to.

We also prove a relatively simple lemma about this new MonoOrder, which states that if $x \leq y$ and $y \leq x$, x must be equal to y.

```
Require Import Orders.
Module MonoOrder <: TotalLeBool.
  Definition t := mono.
  Definition leb m n :=
    match mono_cmp m n with
    | Lt ⇒ true
    | Eq ⇒ true
    | Gt ⇒ false
    end.
  Infix "<=m" := leb (at level 35).
  Lemma leb_total : ∀ m n, (m ≤m n = true) ∨ (n ≤m m = true).
  Proof.
```

```
    intros n m. unfold "<=m". destruct (mono_cmp n m) eqn:Hcomp; auto.
    unfold mono_cmp in *. apply lex_rev_lt_gt in Hcomp. rewrite Hcomp. auto.
  Qed.
End MonoOrder.

Lemma leb_both_eq : ∀ x y,
  is_true (MonoOrder.leb x y) →
  is_true (MonoOrder.leb y x) →
  x = y.
Proof.
  intros x y H H0. unfold is_true, MonoOrder.leb in *.
  destruct (mono_cmp y x) eqn:Hyx; destruct (mono_cmp x y) eqn:Hxy;
  unfold mono_cmp in *;
  try (apply lex_rev_lt_gt in Hxy; rewrite Hxy in Hyx; inversion Hyx);
  try (apply lex_rev_lt_gt in Hyx; rewrite Hxy in Hyx; inversion Hyx);
  try inversion H; try inversion H0.
  apply lex_eq in Hxy; auto.
Qed.
```

After this order has been defined and its totality has been proven, we simply define a new MonoSort module to be a sort based on this MonoOrder.

Now, we have a simple sort function for both monomials and polynomials, as well as a few useful lemmas about the sort functions' correctness.

```
Module Import MonoSort := Sort MonoOrder.
```

One technique that helps us deal with the difficulty of sorted lists is proving that each of our four comparators - lt, *VarOrder*, mono_lt, and MonoOrder - are all transitive. This allows us to seamlessly pass between the standard library's Sorted and **StronglySorted** propositions, making many proofs significantly easier.

All four of these are proved relatively easily, mostly by induction and destructing the comparison of the individual values.

```
Lemma lt_Transitive :
  Relations_1.Transitive lt.
Proof.
  unfold Relations_1.Transitive. intros. apply lt_trans with (m:=y); auto.
Qed.

Lemma VarOrder_Transitive :
  Relations_1.Transitive (fun x y : nat ⇒ is_true (NatOrder.leb x y)).
Proof.
  unfold Relations_1.Transitive, is_true.
  induction x, y, z; intros; try reflexivity; simpl in *.
  - inversion H.
  - inversion H.
  - inversion H0.
```

```
  - apply IHx with (y:=y); auto.
Qed.

Lemma mono_lt_Transitive : Relations_1.Transitive mono_lt.
Proof.
  unfold Relations_1.Transitive, is_true, mono_lt, mono_cmp.
  induction x, y, z; intros; try reflexivity; simpl in *.
  - inversion H.
  - inversion H0.
  - inversion H0.
  - inversion H.
  - inversion H0.
  - destruct (a ?= n0) eqn:Han0.
    + apply compare_eq_iff in Han0. rewrite Han0 in H. destruct (n ?= n0) eqn:Hn0.
      × rewrite compare_antisym in Hn0. unfold CompOpp in Hn0.
        destruct (n0?=n); try inversion Hn0. apply (IHx _ _ H H0).
      × rewrite compare_antisym in Hn0. unfold CompOpp in Hn0.
        destruct (n0?=n); try inversion Hn0. inversion H.
      × inversion H0.
    + auto.
    + destruct (n ?= n0) eqn:Hnn0.
      × apply compare_eq_iff in Hnn0. rewrite Hnn0 in H. rewrite Han0 in H.
        inversion H.
      × apply compare_lt_iff in Hnn0. apply compare_gt_iff in Han0.
        apply lt_trans with (n:=n) in Han0; auto. apply compare_lt_iff in Han0.
        rewrite compare_antisym in Han0. unfold CompOpp in Han0.
        destruct (a?=n); try inversion Han0. inversion H.
      × inversion H0.
Qed.

Lemma MonoOrder_Transitive :
  Relations_1.Transitive (fun x y : list nat ⇒ is_true (MonoOrder.leb x y)).
Proof.
  unfold Relations_1.Transitive, is_true, MonoOrder.leb, mono_cmp.
  induction x, y, z; intros; try reflexivity; simpl in *.
  - inversion H.
  - inversion H.
  - inversion H0.
  - destruct (a ?= n) eqn:Han.
    + apply compare_eq_iff in Han. rewrite Han. destruct (n ?= n0) eqn:Hn0.
      × apply (IHx _ _ H H0).
      × reflexivity.
      × inversion H0.
    + destruct (n ?= n0) eqn:Hn0.
```

```
    × apply compare_eq_iff in Hn0. rewrite ← Hn0. rewrite Han. reflexivity.
    × apply compare_lt_iff in Han. apply compare_lt_iff in Hn0.
      apply (lt_trans a n n0 Han) in Hn0. apply compare_lt_iff in Hn0.
      rewrite Hn0. reflexivity.
    × inversion H0.
  + inversion H.
Qed.
```

### 6.2.2 Sorting and Permutations

The entire purpose of ensuring our monomials and polynomials remain sorted at all times is so that two polynomials containing the same elements are treated as equal. This definition obviously lends itself very well to the use of the Permutation predicate from the standard library, which explains we we proved so many lemmas about permutations during list_util.

When comparing equality of polynomials or monomials, this sort function is often extremely tricky to deal with. Induction over a list being passed to sort is nearly impossible, because the induction element $a$ is not guaranteed to be the least value, so will not easily make it outside of the sort function. As a result, the induction hypothesis is almost always useless.

To combat this, we will prove a series of lemmas relating sort to Permutation, since clearly sorting has no effect when we are comparing the lists in an unordered fashion. The simplest of these lemmas is that if either term of a Permutation is wrapped in a sort function, we can easily get rid of it without changing the provability of these statements.

```
Lemma Permutation_VarSort_l : ∀ m n,
  Permutation m n ↔ Permutation (VarSort.sort m) n.
Proof.
  intros m n. split; intro.
  - apply Permutation_trans with (l':=m). apply Permutation_sym.
    apply VarSort.Permuted_sort. apply H.
  - apply Permutation_trans with (l':=(VarSort.sort m)).
    apply VarSort.Permuted_sort. apply H.
Qed.

Lemma Permutation_VarSort_r : ∀ m n,
  Permutation m n ↔ Permutation m (VarSort.sort n).
Proof.
  intros m n. split; intro.
  - apply Permutation_sym. rewrite ← Permutation_VarSort_l.
    apply Permutation_sym; auto.
  - apply Permutation_sym. rewrite → Permutation_VarSort_l.
    apply Permutation_sym; auto.
Qed.

Lemma Permutation_MonoSort_r : ∀ p q,
```

```
  Permutation p q ↔ Permutation p (sort q).
Proof.
  intros p q. split; intro H.
  - apply Permutation_trans with (l':=q). apply H. apply Permuted_sort.
  - apply Permutation_trans with (l':=(sort q)). apply H. apply Permutation_sym.
    apply Permuted_sort.
Qed.

Lemma Permutation_MonoSort_l : ∀ p q,
  Permutation p q ↔ Permutation (sort p) q.
Proof.
  intros p q. split; intro H.
  - apply Permutation_sym. rewrite ← Permutation_MonoSort_r.
    apply Permutation_sym. auto.
  - apply Permutation_sym. rewrite Permutation_MonoSort_r.
    apply Permutation_sym. auto.
Qed.
```

More powerful is the idea that, if we know we are dealing with sorted lists, there is no difference between proving lists are equal and proving they are Permutations. While this seems intuitive, it is actually fairly complicated to prove in Coq.

For monomials, the proof begins by performing induction on both lists. The first three cases are very straightforward, and the only challenge comes from the third case. We approach the third case by first comparing the two induction elements, $a$ and $a0$.

This forms three goals for us - one where $a = a0$, one where $a < a0$, and one where $a > a0$. The first goal is extremely straightforward, and follows from the induction hypothesis almost immediately after using a few compare lemmas.

This leaves us with the next two goals, which seem to be more challenging at first. However, some further thought leads us to the conclusion that both goals should both be contradictions. If the lists are both sorted, and they contain all the same elements, then they should have the same element, at the head of the list, which is the least element of the set. This element is clearly $a$ for the first list, and $a0$ for the second. However, our destruct of compare has left us with a hypothesis stating that they are not equal! This is the source of the contradiction.

To get Coq to see our contradiction, we first make use of the **Transitive** lemmas we proved earlier to convert to **StronglySorted**. This allows us to get a hypothesis in the second goal that states that $a0$ must be less than everything in the second list. Because $a$ is not equal to $a0$, this implied that $a$ is somewhere else in the second list, and therefore $a0$ is less than $a$. This clearly contradicts the fact that $a < a0$. The third goal looks the same, but in reverse.

```
Lemma Permutation_Sorted_mono_eq : ∀ (m n : mono),
  Permutation m n →
  Sorted (fun n m ⇒ is_true (leb n m)) m →
  Sorted (fun n m ⇒ is_true (leb n m)) n →
  m = n.
```

```
Proof.
  intros m n Hp Hsl Hsm. generalize dependent n.
  induction m; induction n; intros.
  - reflexivity.
  - apply Permutation_nil in Hp. auto.
  - apply Permutation_sym, Permutation_nil in Hp. auto.
  - clear IHn. apply Permutation_incl in Hp as Hp'. destruct Hp'.
    destruct (a ?= a0) eqn:Hcomp.
    + apply compare_eq_iff in Hcomp. rewrite Hcomp in *.
      apply Permutation_cons_inv in Hp. f_equal; auto.
      apply IHm.
      × apply Sorted_inv in Hsl. apply Hsl.
      × apply Hp.
      × apply Sorted_inv in Hsm. apply Hsm.
    + apply compare_lt_iff in Hcomp as Hneq. apply incl_cons_inv in H. destruct H.
      apply Sorted_StronglySorted in Hsm. apply StronglySorted_inv in Hsm as [].
      × simpl in H. destruct H; try (rewrite H in Hneq; apply lt_irrefl in Hneq;
contradiction).
        pose (Forall_In _ _ _ _ H H3). simpl in i. unfold is_true in i.
        apply leb_le in i. apply lt_not_le in Hneq. contradiction.
      × apply VarOrder_Transitive.
    + apply compare_gt_iff in Hcomp as Hneq. apply incl_cons_inv in H0. destruct H0.
      apply Sorted_StronglySorted in Hsl. apply StronglySorted_inv in Hsl as [].
      × simpl in H0. destruct H0; try (rewrite H0 in Hneq; apply gt_irrefl in Hneq;
contradiction).
        pose (Forall_In _ _ _ _ H0 H3). simpl in i. unfold is_true in i.
        apply leb_le in i. apply lt_not_le in Hneq. contradiction.
      × apply VarOrder_Transitive.
Qed.
```

We also wish to prove the same thing for polynomials. This proof is identical in spirit, as we do the same double induction, destructing of compare, and find the same two contradictions. The only difference is the use of lemmas about lex instead of compare, since now we are dealing with lists of lists.

```
Lemma Permutation_Sorted_eq : ∀ (l m : list mono),
  Permutation l m →
  Sorted (fun x y ⇒ is_true (MonoOrder.leb x y)) l →
  Sorted (fun x y ⇒ is_true (MonoOrder.leb x y)) m →
  l = m.
Proof.
  intros l m Hp Hsl Hsm. generalize dependent m.
  induction l; induction m; intros.
  - reflexivity.
```

```
- apply Permutation_nil in Hp. auto.
- apply Permutation_sym, Permutation_nil in Hp. auto.
- clear IHm. apply Permutation_incl in Hp as Hp'. destruct Hp'.
  destruct (mono_cmp a a0) eqn:Hcomp.
  + apply lex_eq in Hcomp. rewrite Hcomp in *.
    apply Permutation_cons_inv in Hp. f_equal; auto.
    apply IHl.
    × apply Sorted_inv in Hsl. apply Hsl.
    × apply Hp.
    × apply Sorted_inv in Hsm. apply Hsm.
  + apply lex_neq' in Hcomp as Hneq. apply incl_cons_inv in H. destruct H.
    apply Sorted_StronglySorted in Hsm. apply StronglySorted_inv in Hsm as [].
    × simpl in H. destruct H; try (rewrite H in Hneq; contradiction).
      pose (Forall_In _ _ _ _ H H3). simpl in i. unfold is_true,
      MonoOrder.leb, mono_cmp in i. apply lex_rev_lt_gt in Hcomp.
      rewrite Hcomp in i. inversion i.
    × apply MonoOrder_Transitive.
  + apply lex_neq' in Hcomp as Hneq. apply incl_cons_inv in H0. destruct H0.
    apply Sorted_StronglySorted in Hsl. apply StronglySorted_inv in Hsl as [].
    × simpl in H0. destruct H0; try (rewrite H0 in Hneq; contradiction).
      pose (Forall_In _ _ _ _ H0 H3). simpl in i. unfold is_true in i.
      unfold MonoOrder.leb in i. rewrite Hcomp in i. inversion i.
    × apply MonoOrder_Transitive.
Qed.
```

Another useful form of these two lemmas is that if at any point we are attempting to prove that sort of one list equals sort of another, we can ditch the sort and instead prove that the two lists are Permutations. These lemmaa will come up a lot in future proofs, and has made some of our work much easier.

```
Lemma Permutation_sort_mono_eq : ∀ (l m:mono),
  Permutation l m ↔ VarSort.sort l = VarSort.sort m.
Proof.
  intros l m. split; intros H.
  - assert (H0 : Permutation (VarSort.sort l) (VarSort.sort m)).
    + apply Permutation_trans with (l:=(VarSort.sort l)) (l':=m) (l'':=(VarSort.sort m)).
      × apply Permutation_sym. apply Permutation_sym in H.
        apply (Permutation_trans H (VarSort.Permuted_sort l)).
      × apply VarSort.Permuted_sort.
    + apply (Permutation_Sorted_mono_eq _ _ H0 (VarSort.LocallySorted_sort l) (VarSort.LocallySorted_sort
m)).
  - assert (Permutation (VarSort.sort l) (VarSort.sort m)).
    + rewrite H. apply Permutation_refl.
    + pose (VarSort.Permuted_sort l). pose (VarSort.Permuted_sort m).
```

```
    apply (Permutation_trans p) in H0. apply Permutation_sym in p0.
    apply (Permutation_trans H0) in p0. apply p0.
Qed.

Lemma Permutation_sort_eq : ∀ l m,
  Permutation l m ↔ sort l = sort m.
Proof.
  intros l m. split; intros H.
  - assert (H0 : Permutation (sort l) (sort m)).
    + apply Permutation_trans with (l:=(sort l)) (l':=m) (l'':=(sort m)).
      × apply Permutation_sym. apply Permutation_sym in H.
        apply (Permutation_trans H (Permuted_sort l)).
      × apply Permuted_sort.
    + apply (Permutation_Sorted_eq _ _ H0 (LocallySorted_sort l) (LocallySorted_sort m)).
  - assert (Permutation (sort l) (sort m)).
    + rewrite H. apply Permutation_refl.
    + pose (Permuted_sort l). pose (Permuted_sort m).
      apply (Permutation_trans p) in H0. apply Permutation_sym in p0.
      apply (Permutation_trans H0) in p0. apply p0.
Qed.
```

## 6.3 Repairing Invalid Monomials & Polynomials

Clearly, there is a very strict set of rules we would like to be true about all of the polynomials and monomials we workd with. These rules are, however, relatively tricky to maintain when it comes to writing functions that operate over monomials and polynomials. Rather than rely on our ability to define every function to perfectly maintain this set of rules, we decided to define two functions to "repair" any invalid monomials or polynomials. These functions, given a list of variables or a list of list of variables, will apply a few functions to them such that at the end, we are left with a properly formatted monomial or polynomial.

### 6.3.1 Converting Between lt and *le*

A small problem with the sort function provided by the standard library is that it requires us to use a *le* comparator, as opposed to lt like we use in our is_mono and is_poly definitions. However, as we said before, because our lists have no duplicates *le* and lt are equivalent. Obviously, though, saying this isn't enough - we must prove it for it to be useful to us in proofs.

The first step to proving this is proving that this is true when dealing with the **HdRel** definition that Sorted is built on top of. These lemmas state that, if $a$ holds the *le* relation with a list, and there are also no duplicates in $a::l$, that $a$ also holds the lt relation with the list. These proofs are both relatively straightforward, especially with the use of the NoDup_neq lemma proven earlier.

```
Lemma HdRel_le_lt : ∀ a m,
  HdRel (fun n m ⇒ is_true (leb n m)) a m ∧ NoDup (a::m) → HdRel lt a m.
Proof.
  intros a m []. remember (fun n m ⇒ is_true (leb n m)) as le.
  destruct m.
  - apply HdRel_nil.
  - apply HdRel_cons. apply HdRel_inv in H.
    apply (NoDup_neq _ a n) in H0; intuition. rewrite Heqle in H.
    unfold is_true in H. apply leb_le in H. destruct (a ?= n) eqn:Hcomp.
    + apply compare_eq_iff in Hcomp. contradiction.
    + apply compare_lt_iff in Hcomp. apply Hcomp.
    + apply compare_gt_iff in Hcomp. apply leb_correct_conv in Hcomp.
      apply leb_correct in H. rewrite H in Hcomp. inversion Hcomp.
Qed.

Lemma HdRel_mono_le_lt : ∀ a p,
  HdRel (fun n m ⇒ is_true (MonoOrder.leb n m)) a p ∧ NoDup (a::p) →
  HdRel mono_lt a p.
Proof.
  intros a p []. remember (fun n m ⇒ is_true (MonoOrder.leb n m)) as le.
  destruct p.
  - apply HdRel_nil.
  - apply HdRel_cons. apply HdRel_inv in H.
    apply (NoDup_neq _ a l) in H0; intuition. rewrite Heqle in H.
    unfold is_true in H. unfold MonoOrder.leb in H. unfold mono_lt.
    destruct (mono_cmp a l) eqn:Hcomp.
    + apply lex_eq in Hcomp. contradiction.
    + reflexivity.
    + inversion H.
Qed.
```

Now, to apply these lemmas - we prove that if a list is Sorted with a *le* operator and has no duplicates, that it is also Sorted with the corresponding lt operator.

```
Lemma VarSort_Sorted : ∀ (m : mono),
  Sorted (fun n m ⇒ is_true (leb n m)) m ∧ NoDup m → Sorted lt m.
Proof.
  intros m []. remember (fun n m ⇒ is_true (leb n m)) as le.
  induction m.
  - apply Sorted_nil.
  - apply Sorted_inv in H. apply Sorted_cons.
    + apply IHm.
      × apply H.
      × apply NoDup_cons_iff in H0. apply H0.
    + apply HdRel_le_lt. split.
```

× rewrite ← *Heqle*. apply *H*.
× apply *H0*.
Qed.

Lemma MonoSort_Sorted : ∀ ($p$ : poly),
  **Sorted** (fun $n$ $m$ ⇒ is_true (MonoOrder.leb $n$ $m$)) $p$ ∧ **NoDup** $p$ →
  **Sorted** mono_lt $p$.
Proof.
  intros $p$ []. *remember* (fun $n$ $m$ ⇒ is_true (MonoOrder.leb $n$ $m$)) as *le*.
  induction $p$.
  - apply Sorted_nil.
  - apply Sorted_inv in *H*. apply Sorted_cons.
    + apply *IHp*.
      × apply *H*.
      × apply NoDup_cons_iff in *H0*. apply *H0*.
    + apply HdRel_mono_le_lt. split.
      × rewrite ← *Heqle*. apply *H*.
      × apply *H0*.
Qed.

For convenience, we also include the inverse - if a list is Sorted with an lt operator, it is also Sorted with the matching *le* operator.

Lemma Sorted_VarSorted : ∀ ($m$ : mono),
  **Sorted** lt $m$ →
  **Sorted** (fun $n$ $m$ ⇒ is_true (leb $n$ $m$)) $m$.
Proof.
  intros $m$ *H*. induction *H*.
  - apply Sorted_nil.
  - apply Sorted_cons.
    + apply *IHSorted*.
    + destruct $l$.
      × apply HdRel_nil.
      × apply HdRel_cons. apply HdRel_inv in *H0*. apply lt_le_incl in *H0*.
        apply leb_le in *H0*. apply *H0*.
Qed.

Lemma Sorted_MonoSorted : ∀ ($p$ : poly),
  **Sorted** mono_lt $p$ →
  **Sorted** (fun $n$ $m$ ⇒ is_true (MonoOrder.leb $n$ $m$)) $p$.
Proof.
  intros $p$ *H*. induction *H*.
  - apply Sorted_nil.
  - apply Sorted_cons.
    + apply *IHSorted*.

```
    + destruct l.
      × apply HdRel_nil.
      × apply HdRel_cons. apply HdRel_inv in H0. unfold MonoOrder.leb.
        rewrite H0. auto.
Qed.
```

Another obvious side effect of what we have just proven is that if a list is Sorted with an lt operator, clearly there are no duplicates, as no elements are equal to each other.

```
Lemma NoDup_MonoSorted : ∀ (p : poly),
  Sorted mono_lt p →
  NoDup p.
Proof.
  intros p H. apply Sorted_StronglySorted in H.
  - induction p.
    + auto.
    + apply StronglySorted_inv in H as []. apply NoDup_forall_neq.
      × apply Forall_forall. intros x Hin. rewrite Forall_forall in H0.
        pose (lex_neq' a x). destruct a0. apply H1 in H0; auto.
      × apply IHp. apply H.
  - apply mono_lt_Transitive.
Qed.
```

```
Lemma NoDup_VarSorted : ∀ (m : mono),
  Sorted lt m → NoDup m.
Proof.
  intros p H. apply Sorted_StronglySorted in H.
  - induction p.
    + auto.
    + apply StronglySorted_inv in H as []. apply NoDup_forall_neq.
      × apply Forall_forall. intros x Hin. rewrite Forall_forall in H0.
        apply lt_neq. apply H0. apply Hin.
      × apply IHp. apply H.
  - apply lt_Transitive.
Qed.
```

There are a few more useful lemmas we would like to prove about our sort functions before we can define and prove the correctness of our repair functions. Mostly, we want to know that sorting a list has no effect on some properties of it.

Specifically, if an element was In a list before it was sorted, it is also in it after, and vice versa. Similarly, if a list has no duplicates before being sorted, it also has no duplicates after.

```
Lemma In_sorted : ∀ a l,
  In a l ↔ In a (sort l).
Proof.
```

```
  intros a l. pose (MonoSort.Permuted_sort l). split; intros Hin.
  - apply (Permutation_in _ p Hin).
  - apply (Permutation_in' (Logic.eq_refl a) p). auto.
Qed.

Lemma NoDup_VarSort : ∀ (m : mono),
  NoDup m → NoDup (VarSort.sort m).
Proof.
  intros m Hdup. pose (VarSort.Permuted_sort m).
  apply (Permutation_NoDup p Hdup).
Qed.

Lemma NoDup_MonoSort : ∀ (p : poly),
  NoDup p → NoDup (MonoSort.sort p).
Proof.
  intros p Hdup. pose (MonoSort.Permuted_sort p).
  apply (Permutation_NoDup p0 Hdup).
Qed.
```

### 6.3.2 Defining the Repair Functions

Now time for our definitions. To convert a list of variables into a monomial, we first apply nodup, which removes all duplicates. We use nodup rather than nodup_cancel because $x \times x = x$, so we want one copy to remain. After applying nodup, we use our VarSort module to sort the list from least to greatest.

```
Definition make_mono (l : list nat) : mono :=
  VarSort.sort (nodup var_eq_dec l).
```

The process of converting a list of list of variables into a polynomial is very similar. First we map across the list applying make_mono, so that each sublist is properly formatted. Then we apply nodup_cancel to remove duplicates. In this case, we use nodup_cancel instead of nodup because $x+x = 0$, so we want pairs to cancel out. Lastly, we use our MonoSort module to sort the list.

```
Definition make_poly (l : list mono) : poly :=
  MonoSort.sort (nodup_cancel mono_eq_dec (map make_mono l)).

Lemma make_poly_refold : ∀ p,
  sort (nodup_cancel mono_eq_dec (map make_mono p)) =
  make_poly p.
Proof. auto. Qed.
```

Now to prove the correctness of these lists - if you apply make_mono to something, it is then guaranteed to satisfy the is_mono proposition. This proof is relatively straightforward, as we have already done most of the work with VarSort_Sorted; all that is left to do is show that make_mono $m$ is Sorted and has NoDups, which is obvious considering that is exactly what make_mono does!

```
Lemma make_mono_is_mono : ∀ m,
  is_mono (make_mono m).
Proof.
  intros m. unfold is_mono, make_mono. apply VarSort_Sorted. split.
  + apply VarSort.LocallySorted_sort.
  + apply NoDup_VarSort. apply NoDup_nodup.
Qed.
```

The proof for make_poly_is_poly is almost identical, with the addition of one part. is_poly still asks us to prove that the list is Sorted, which follows from MonoSort_Sorted like above. The only difference is that is_poly also asks us to show that each element in the list is_mono, which follows from the use of a few In lemmas and the make_mono_is_mono we just proved thanks to the map in make_poly.

```
Lemma make_poly_is_poly : ∀ p,
  is_poly (make_poly p).
Proof.
  intros p. unfold is_poly, make_poly. split.
  - apply MonoSort_Sorted. split.
    + apply MonoSort.LocallySorted_sort.
    + apply NoDup_MonoSort. apply NoDup_nodup_cancel.
  - intros m Hm. apply In_sorted in Hm. apply nodup_cancel_in in Hm.
    apply in_map_iff in Hm. destruct Hm. destruct H. rewrite ← H.
    apply make_mono_is_mono.
Qed.

Hint Resolve make_poly_is_poly make_mono_is_mono.
```

### 6.3.3 Facts about make_mono

Before we dive into more complicated proofs involving these repair functions, there are a few simple lemmas we can prove about them.

First is that if some variable $x$ was in a list before make_mono was applied, it must also be in it after, and vice-versa.

```
Lemma make_mono_In : ∀ x m,
  In x (make_mono m) ↔ In x m.
Proof.
  intros x m. split; intro H.
  - unfold make_mono in H. pose (VarSort.Permuted_sort (nodup var_eq_dec m)).
    apply Permutation_sym in p. apply (Permutation_in _ p) in H. apply nodup_In in H.
auto.
  - unfold make_mono. pose (VarSort.Permuted_sort (nodup var_eq_dec m)).
    apply Permutation_in with (l:=(nodup var_eq_dec m)); auto. apply nodup_In. auto.
Qed.
```

In addition, if some list $m$ is already a monomial, removing anything from it will not change that.

```
Lemma remove_is_mono : ∀ x m,
  is_mono m →
  is_mono (remove var_eq_dec x m).
Proof.
  intros x m H. unfold is_mono in *. apply StronglySorted_Sorted.
  apply StronglySorted_remove. apply Sorted_StronglySorted in H. auto.
  apply lt_Transitive.
Qed.
```

If we know that some (l1++x::l2) is a mono, then clearly it is still a monomial if we remove the x from the middle, as this will not affect the sorting at all.

```
Lemma mono_middle : ∀ x l1 l2,
  is_mono (l1 ++ x :: l2) →
  is_mono (l1 ++ l2).
Proof.
  intros x l1 l2 H. unfold is_mono in *. apply Sorted_StronglySorted in H.
  apply StronglySorted_Sorted. induction l1.
  - rewrite app_nil_l in *. apply StronglySorted_inv in H as []; auto.
  - simpl in *. apply StronglySorted_inv in H as []. apply SSorted_cons; auto.
    apply Forall_forall. rewrite Forall_forall in H0. intros x0 Hin.
    apply H0. apply in_app_iff in Hin as []; intuition.
  - apply lt_Transitive.
Qed.
```

Due to the nature of sorting, the order of arguments in make_mono doesn't matter.

```
Lemma make_mono_app_comm : ∀ m n,
  make_mono (m ++ n) = make_mono (n ++ m).
Proof.
  intros m n. apply Permutation_sort_mono_eq. apply Permutation_nodup.
  apply Permutation_app_comm.
Qed.
```

Finally, is a list m is a member of the list resulting from map make_mono, then clearly it is a monomial.

```
Lemma mono_in_map_make_mono : ∀ p m,
  In m (map make_mono p) → is_mono m.
Proof.
  intros. apply in_map_iff in H as [x []]. rewrite ← H. auto.
Qed.
```

### 6.3.4 Facts about make_poly

If two lists are permutations of each other, then they will be equivalent after applying make_poly to both.

```
Lemma make_poly_Permutation : ∀ p q,
  Permutation p q → make_poly p = make_poly q.
Proof.
  intros. unfold make_poly.
  apply Permutation_sort_eq, nodup_cancel_Permutation, Permutation_map.
  auto.
Qed.
```

Because we have shown that sort and Permutation are equivalent, we can easily show that the order of elements in a call to make_poly does not matter.

```
Lemma make_poly_app_comm : ∀ p q,
  make_poly (p ++ q) = make_poly (q ++ p).
Proof.
  intros p q. apply Permutation_sort_eq.
  apply nodup_cancel_Permutation. apply Permutation_map.
  apply Permutation_app_comm.
Qed.
```

During make_poly, we both sort and call nodup_cancel. A lemma that is useful in some cases shows that it doesn't matter what order we do these in, as nodup_cancel will maintain the order of a list.

```
Lemma sort_nodup_cancel_assoc : ∀ l,
  sort (nodup_cancel mono_eq_dec l) = nodup_cancel mono_eq_dec (sort l).
Proof.
  intros l. apply Permutation_Sorted_eq.
  - pose (Permuted_sort (nodup_cancel mono_eq_dec l)). apply Permutation_sym in p.
    apply (Permutation_trans p). clear p. apply NoDup_Permutation.
    + apply NoDup_nodup_cancel.
    + apply NoDup_nodup_cancel.
    + intros x. split.
      × intros H. apply Permutation_in with (l:=(nodup_cancel mono_eq_dec l)).
        apply nodup_cancel_Permutation. apply Permuted_sort. auto.
      × intros H. apply Permutation_in with (l:=(nodup_cancel mono_eq_dec (sort l))).
        apply nodup_cancel_Permutation. apply Permutation_sym. apply Permuted_sort.
auto.
  - apply LocallySorted_sort.
  - apply Sorted_nodup_cancel.
    + apply MonoOrder_Transitive.
    + apply LocallySorted_sort.
Qed.
```

Another obvious but useful lemma is that if a monomial $m$ is in a list resulting from applying make_poly, is is clearly a monomial.

```
Lemma mono_in_make_poly : ∀ p m,
  In m (make_poly p) → is_mono m.
Proof.
  intros. unfold make_poly in H. apply In_sorted in H.
  apply nodup_cancel_in in H. apply (mono_in_map_make_mono _ _ H).
Qed.
```

## 6.4 Proving Functions "Pointless"

In the list_util file, we have two lemmas revolving around the idea that, in some cases, calling nodup_cancel is "pointless". The idea here is that, when comparing very complicated terms, it is sometimes beneficial to either add or remove an extra function call that has no effect on the final term. Until this point, we have only proven this about nodup_cancel, but there are many other cases where this is true, which will make our more complex proofs much easier. This section serves to prove this true of most of our functions.

### 6.4.1 Working with sort Functions

The next two lemmas very simply prove that, if a list is already Sorted, then calling either VarSort or MonoSort on it will have no effect. This is relatively obvious, and is extremely easy to prove with our Permutation / Sorted lemmas from earlier.

```
Lemma no_sort_VarSorted : ∀ m,
  Sorted lt m →
  VarSort.sort m = m.
Proof.
  intros m H. apply Permutation_Sorted_mono_eq.
  - apply Permutation_sym. apply VarSort.Permuted_sort.
  - apply VarSort.LocallySorted_sort.
  - apply Sorted_VarSorted. auto.
Qed.
```

```
Lemma no_sort_MonoSorted : ∀ p,
  Sorted mono_lt p →
  MonoSort.sort p = p.
Proof.
  intros p H. unfold make_poly. apply Permutation_Sorted_eq.
  - apply Permutation_sym. apply Permuted_sort.
  - apply LocallySorted_sort.
  - apply Sorted_MonoSorted. auto.
Qed.
```

The following lemma more closely aligns with the format of the nodup_cancel_pointless lemma from list_util. It states that if the result of appending two lists is already going to be sorted, there is no need to sort the intermediate lists.

This also applies if the sort is wrapped around the right argument, thanks to the Permutation lemmas we proved earlier.

```
Lemma sort_pointless : ∀ p q,
  sort (sort p ++ q) =
  sort (p ++ q).
Proof.
  intros p q. apply Permutation_sort_eq.
  apply Permutation_app_tail. apply Permutation_sym.
  apply Permuted_sort.
Qed.
```

### 6.4.2 Working with make_mono

There are a couple forms that the proof of make_mono being pointless can take. Firstly, because we already know that make_mono simply applies functions to get the list into a form that satisfies is_mono, it makes sense to prove that if some list is already a mono that make_mono will have no effect. This is proved with the help of no_sort_VarSorted and no_nodup_NoDup.

```
Lemma no_make_mono : ∀ m,
  is_mono m →
  make_mono m = m.
Proof.
  unfold make_mono, is_mono. intros m H. rewrite no_sort_VarSorted.
  - apply no_nodup_NoDup. apply NoDup_VarSorted in H. auto.
  - apply Sorted_nodup.
    + apply lt_Transitive.
    + auto.
Qed.
```

We can also prove the more standard form of make_mono_pointless, which states that if there are nested calls to make_mono, we can remove all except the outermost layer.

```
Lemma make_mono_pointless : ∀ m a,
  make_mono (m ++ make_mono a) = make_mono (m ++ a).
Proof.
  intros m a. apply Permutation_sort_mono_eq. rewrite ← (nodup_pointless _ a).
  apply Permutation_nodup. apply Permutation_app_head. unfold make_mono.
  rewrite ← Permutation_VarSort_l. auto.
Qed.
```

Similarly, if we already know that all of the elements in a list are monos, then mapping make_mono across the list will have no effect on the entire list.

```
Lemma no_map_make_mono : ∀ p,
  (∀ m, In m p → is_mono m) →
  map make_mono p = p.
Proof.
  intros p H. induction p.
  - auto.
  - simpl. rewrite no_make_mono.
    + f_equal. apply IHp. intros m Hin. apply H. intuition.
    + apply H. intuition.
Qed.
```

Lastly, the pointless proof that more closely aligns with what we have done so far - if make_poly is already being applied to a list, there is no need to have a call to map make_mono on the inside.

```
Lemma map_make_mono_pointless : ∀ p q,
  make_poly (map make_mono p ++ q) =
  make_poly (p ++ q).
Proof.
  intros p q. destruct p.
  - auto.
  - simpl. unfold make_poly. simpl map. rewrite (no_make_mono (make_mono l)); auto.
    rewrite map_app. rewrite map_app. rewrite (no_map_make_mono (map _ _)).
    auto. intros m Hin. apply in_map_iff in Hin. destruct Hin as [x[]].
    rewrite ← H. auto.
Qed.
```

### 6.4.3 Working with make_poly

Finally, we work to prove some lemmas about make_poly as a whole being pointless. These proofs are built upon the previous few lemmas, which prove that we can remove the components of make_poly one by one.

First up, we have a lemma that shows that if $p$ already has no duplicates and everything in the list is a mono, then nodup_cancel and map make_mono will both have no effect. This lemma turns out to be very useful *after* something like Permutation_sort_eq has been applied, as it can strip away the other two functions of make_poly.

```
Lemma unsorted_poly : ∀ p,
  NoDup p →
  (∀ m, In m p → is_mono m) →
  nodup_cancel mono_eq_dec (map make_mono p) = p.
Proof.
```

```
  intros p Hdup Hin. rewrite no_map_make_mono; auto.
  apply no_nodup_cancel_NoDup; auto.
Qed.
```

Similarly to no_make_mono, it is very straightforward to prove that if some list $p$ is already a poly, then make_poly has no effect.

```
Lemma no_make_poly : ∀ p,
  is_poly p →
  make_poly p = p.
Proof.
  unfold make_poly, is_poly. intros m []. rewrite no_sort_MonoSorted.
  - rewrite no_nodup_cancel_NoDup.
    + apply no_map_make_mono. intros m0 Hin. apply H0. auto.
    + apply NoDup_MonoSorted in H. rewrite no_map_make_mono; auto.
  - apply Sorted_nodup_cancel.
    + apply mono_lt_Transitive.
    + rewrite no_map_make_mono; auto.
Qed.
```

Now onto the most important lemma. In many of the later proofs, there will be times where there are calls to make_poly nested inside of each other, or long lists of arguments appended together inside of a make_poly. In either case, the ability to add and remove extra calls to make_poly as we please proves to be very powerful.

To prove make_poly_pointless, we begin by proving a weaker version that insists that all of the arguments of $p$ and $q$ are all monomials. This addition makes the proof significantly easier. As one might expect, the proof is completed by using Permutation_sort_eq to remove the sort calls, nodup_cancel_pointless to remove the nodup_cancel calls, and no_map_make_mono to get rid of the map make_mono calls. After this is done, the two sides are identical.

```
Lemma make_poly_pointless_weak : ∀ p q,
  (∀ m, In m p → is_mono m) →
  (∀ m, In m q → is_mono m) →
  make_poly (make_poly p ++ q) =
  make_poly (p ++ q).
Proof.
  intros p q Hmp Hmq. unfold make_poly.
  repeat rewrite no_map_make_mono; intuition.
  apply Permutation_sort_eq. rewrite sort_nodup_cancel_assoc.
  rewrite nodup_cancel_pointless. apply nodup_cancel_Permutation.
  apply Permutation_sym. apply Permutation_app_tail. apply Permuted_sort.
  - simpl in H. rewrite in_app_iff in H. destruct H; intuition.
  - rewrite in_app_iff in H. destruct H; intuition.
    apply In_sorted in H. apply nodup_cancel_in in H. intuition.
Qed.
```

Now, to make the stronger and easier to use version, we simply rewrite in the opposite direction with map_make_mono_pointless to add extra calls map make_mono in! Ironically, this proof *of* make_poly_pointless is a great example of why these "pointless" lemmas are so useful. While we can clearly tell that adding the extra call to map make_mono makes no difference, it makes proving things in a way that Coq understands dramatically easier at times.

After rewriting with map_make_mono_pointless, clearly both areguments contain all monomials, and we can use make_poly_pointless_weak to prove the stronger version.

```
Lemma make_poly_pointless : ∀ p q,
  make_poly (make_poly p ++ q) =
  make_poly (p ++ q).
Proof.
  intros p q. rewrite make_poly_app_comm.
  rewrite ← map_make_mono_pointless. rewrite make_poly_app_comm.
  rewrite ← (map_make_mono_pointless p). rewrite (make_poly_app_comm _ q).
  rewrite ← (map_make_mono_pointless q). rewrite (make_poly_app_comm _ (map make_mono
p)).
  rewrite ← (make_poly_pointless_weak (map make_mono p)). unfold make_poly.
  rewrite (no_map_make_mono (map make_mono p)). auto.
  apply mono_in_map_make_mono. apply mono_in_map_make_mono.
  apply mono_in_map_make_mono.
Qed.
```

For convenience, we also prove that it applies on the right side by using make_poly_app_comm twice.

```
Lemma make_poly_pointless_r : ∀ p q,
  make_poly (p ++ make_poly q) =
  make_poly (p ++ q).
Proof.
  intros p q. rewrite make_poly_app_comm. rewrite make_poly_pointless.
  apply make_poly_app_comm.
Qed.
```

## 6.5 Polynomial Arithmetic

Now, the foundation for operations on polynomails has been put in place, and we can begin to get into the real meat - our arithmetic operators. First up is addition. Because we have so cleverly defined our make_poly function, addition over our data structures is as simple as appending the two polynomials and repairing the result back into a proper polynomial.

We also include a simple refold lemma for convenience, and a quick proof that the result of addPP is always a poly.

```
Definition addPP (p q : poly) : poly :=
```

```
  make_poly (p ++ q).

Lemma addPP_refold : ∀ p q,
  make_poly (p ++ q) = addPP p q.
Proof.
  auto.
Qed.

Lemma addPP_is_poly : ∀ p q,
  is_poly (addPP p q).
Proof.
  intros p q. apply make_poly_is_poly.
Qed.
```

Similarly, the definition for multiplication becomes much easier with the creation of make_poly. All we need to do is use our distribute function defined earlier to form all combinations of one monomial from each list, and call make_poly on the result.

```
Definition mulPP (p q : poly) : poly :=
  make_poly (distribute p q).

Lemma mulPP_is_poly : ∀ p q,
  is_poly (mulPP p q).
Proof.
  intros p q. apply make_poly_is_poly.
Qed.
```

While this definition is elegant, sometimes it is hard to work with. This has led us to also create a few more definitions of multiplication. Each is just slightly different from the last, which allows us to choose the level of completeness we need for any given multiplication proof while knowing that at the end of the day, they are all equivalent.

Each of these new definitions breaks down multiplication into two steps - mutliplying a monomial times a polynomial, and multiplying a polynomial times a polynomial. Multiplying a monomial times a polynomial is simply appending the monomial to each monomial in the polynomial, and multiplying two polynomials is just multiplying each monomial in one polynomial times the other polynomial.

The difference in each of the following definitions comes from the intermediate step. Because we know that mulPP will call make_poly, there is no need to call make_poly on the result of mulMP, as shown in the first definition. However, some proofs are made easier if the result of mulMP is wrapped in map make_mono, and some are made easier if the result is wrapped in a full make_poly. As a result, we have created each of these definitions, and choose between them to help make our proofs easier.

We also include a refolding method for each, for convenience, and a proof that each new version is equivalent to the last.

```
Definition mulMP (p : poly) (m : mono) : poly :=
  map (app m) p.
```

```
Definition mulPP' (p q : poly) : poly :=
  make_poly (concat (map (mulMP p) q)).

Lemma mulPP'_refold : ∀ p q,
  make_poly (concat (map (mulMP p) q)) =
  mulPP' p q.
Proof. auto. Qed.

Lemma mulPP_mulPP' : ∀ (p q : poly),
  mulPP p q = mulPP' p q.
Proof.
  intros p q. unfold mulPP, mulPP'. induction q.
  - auto.
  - simpl. unfold distribute. simpl. unfold mulMP. auto.
Qed.
```

Next, the version including a map make_mono:

```
Definition mulMP' (p : poly) (m : mono) : poly :=
  map make_mono (map (app m) p).

Definition mulPP'' (p q : poly) : poly :=
  make_poly (concat (map (mulMP' p) q)).

Lemma mulPP''_refold : ∀ p q,
  make_poly (concat (map (mulMP' p) q)) =
  mulPP'' p q.
Proof. auto. Qed.

Lemma mulPP'_mulPP'' : ∀ p q,
  mulPP' p q = mulPP'' p q.
Proof.
  intros p q. unfold mulPP', mulPP'', mulMP, mulMP', make_poly.
  rewrite concat_map_map.
  rewrite (no_map_make_mono (map _ _)); auto.
  intros. apply in_map_iff in H as [n []].
  rewrite ← H.
  auto.
Qed.
```

And finally, the version including a full make_poly:

```
Definition mulMP'' (p : poly) (m : mono) : poly :=
  make_poly (map (app m) p).

Definition mulPP''' (p q : poly) : poly :=
  make_poly (concat (map (mulMP'' p) q)).

Lemma mulPP'''_refold : ∀ p q,
  make_poly (concat (map (mulMP'' p) q)) =
```

```
  mulPP''' p q.
Proof. auto. Qed.
```

In order to make the proof of going from mulPP'' to mulPP''' easier, we begin by proving that we can go from their corresponding mulMPs if they are wrapped in a make_poly.

```
Lemma mulMP'_mulMP'' : ∀ m p q,
  make_poly (mulMP' p m ++ q) = make_poly (mulMP'' p m ++ q).
Proof.
  intros m p q. unfold mulMP', mulMP''. rewrite make_poly_app_comm.
  rewrite ← map_make_mono_pointless. rewrite make_poly_app_comm.
  rewrite ← make_poly_pointless. unfold make_poly at 2. rewrite (no_map_make_mono
(map make_mono _)).
  unfold make_poly at 3. rewrite (make_poly_app_comm _ q).
  rewrite ← (map_make_mono_pointless q). rewrite make_poly_app_comm. auto.
  apply mono_in_map_make_mono.
Qed.

Lemma mulPP''_mulPP''' : ∀ p q,
  mulPP'' p q = mulPP''' p q.
Proof.
  intros p q. induction q. auto. unfold mulPP'', mulPP'''. simpl.
  rewrite mulMP'_mulMP''. repeat rewrite ← (make_poly_pointless_r _ (concat _)).
  f_equal. f_equal. apply IHq.
Qed.
```

Again, for convenience, we add lemmas to skip from mulPP to any of the other varieties.

```
Lemma mulPP_mulPP'' : ∀ p q,
  mulPP p q = mulPP'' p q.
Proof.
  intros. rewrite mulPP_mulPP', mulPP'_mulPP''. auto.
Qed.

Lemma mulPP_mulPP''' : ∀ p q,
  mulPP p q = mulPP''' p q.
Proof.
  intros. rewrite mulPP_mulPP'', mulPP''_mulPP'''. auto.
Qed.

Hint Unfold addPP mulPP mulPP' mulPP'' mulPP''' mulMP mulMP' mulMP''.
```

## 6.6 Proving the 10 B-Unification Axioms

Now that we have defined our operations so carefully, we want to prove that the 10 standard B-Unification Axioms all apply. This is extremely important, as they will both be needed in the higher-level proofs of our unification algorithm, and they show that our list-of-list setup is actually correct and equivalent to any other representation of a term.

### 6.6.1 Axiom 1: Additive Inverse

We begin with the inverse and identities for each addition and multiplication. First is the additive inverse, which states that forall terms x, $x + x = 0$.

Thanks to the definition of nodup_cancel and the previously proven nodup_cancel_self, this proof is extremely simple.

```
Lemma addPP_p_p : ∀ p,
  addPP p p = [].
Proof.
  intros p. unfold addPP. unfold make_poly. rewrite map_app.
  rewrite nodup_cancel_self. auto.
Qed.
```

### 6.6.2 Axiom 2: Additive Identity

Next, we prove the additive identity: for all terms x, $0 + x = x$. This also applies in the right direction, and is extremely easy to prove since we already know that appending nil to a list results in that list.

Something to note is that, unlike some of the other of the ten axioms, this one is *only* true if $p$ is already a polynomial. Clearly, if it wasn't, addPP would not return the same $p$, but rather make_poly $p$, since addPP will only return proper polynomials.

```
Lemma addPP_0 : ∀ p,
  is_poly p →
  addPP [] p = p.
Proof.
  intros p Hpoly. unfold addPP. simpl. apply no_make_poly. auto.
Qed.
```

```
Lemma addPP_0r : ∀ p,
  is_poly p →
  addPP p [] = p.
Proof.
  intros p Hpoly. unfold addPP. rewrite app_nil_r. apply no_make_poly. auto.
Qed.
```

### 6.6.3 Axiom 3: Multiplicative Identity - 1

Now onto multiplication. In B-Unification, there are *two* multiplicative identities. We begin with the easier to prove of the two, which is 1. In other words, for any term x, $x \times 1 = x$.

This proof is also very simply proved simply because of how appending nil works

```
Lemma mulPP_1r : ∀ p,
  is_poly p →
  mulPP p [[]] = p.
Proof.
  intros p H. unfold mulPP, distribute. simpl. rewrite app_nil_r.
  rewrite map_id. apply no_make_poly. auto.
Qed.
```

### 6.6.4 Axiom 4: Multiplicative Inverse

Next is the multiplicative inverse, which states that for any term x, $0 \times x = 0$.

This is proven immediately by the distribute_nil lemmas we proved in list_util.

```
Lemma mulPP_0 : ∀ p,
  mulPP [] p = [].
Proof.
  intros p. unfold mulPP. rewrite (@distribute_nil var). auto.
Qed.
```

```
Lemma mulPP_0r : ∀ p,
  mulPP p [] = [].
Proof.
  intros p. unfold mulPP. rewrite (@distribute_nil_r var). auto.
Qed.
```

### 6.6.5 Axiom 5: Commutativity of Addition

The next of the ten axioms states that, for all terms x and y, $x + y = y + x$.

This axiom is also rather easy, and follows entirely from the make_poly_app_comm lemma we proved earlier due to our clever addition definition.

```
Lemma addPP_comm : ∀ p q,
  addPP p q = addPP q p.
Proof.
  intros p q. unfold addPP. apply make_poly_app_comm.
Qed.
```

### 6.6.6 Axiom 6: Associativity of Addition

The next axiom states that, for all terms x, y, and z, $x + (y + z) = (x + y) + z$.

Thanks to addPP_comm and all of the "pointless" lemmas we proved earlier, this proof is much easier than it might have been otherwise. These lemmas allow us to easily manipulate the operations until we end by proving that $p \mathbin{++} q \mathbin{++} r$ is a permutation of $q \mathbin{++} r \mathbin{++} p$.

```
Lemma addPP_assoc : ∀ p q r,
```

```
  addPP (addPP p q) r = addPP p (addPP q r).
Proof.
  intros p q r. rewrite (addPP_comm _ (addPP _ _)). unfold addPP.
  repeat rewrite make_poly_pointless. repeat rewrite ← app_assoc.
  apply Permutation_sort_eq. apply nodup_cancel_Permutation. apply Permutation_map.
  rewrite (app_assoc q). apply Permutation_app_comm with (l':=(q++r)).
Qed.
```

### 6.6.7 Axiom 7: Commutativity of Multiplication

Now onto the harder half of the axioms. This next one states that for all terms x and y, $x \times y = y \times x$. In order to prove this, we have opted to use the second version of mulPP, which wraps the monomial multiplication in a map make_mono.

The proof begins with double induction, and the first three cases are rather simple. The fourth case is slightly more complicated, but the make_poly_pointless lemma we proved earlier plays a huge role in making it simpler. We begin by simplifying, so that the $m$ created by induction on $q$ is distributed across the list on the left side, and the $a$ created by induction on $p$ is distributed accross the list on the right side. Then, we use make_poly_pointless to surround the rightmost term - which now has $a$ but not $m$ on the left and $m$ but not $a$ on the right - with make_poly. This additional make_poly allows us to refold the mess of maps and concats into mulPP, like they used to be. From there, we use the two induction hypotheses to apply commutativity, remove the redundant make_polys we added, and simple again.

In this way, we are able to cause both $a$ and $m$ to be distributed across the whole list on both the left and right sides of the equation. At this point, it simply requires some rearranging of app with the help of Permutation, and our left and right sides are equal.

Without the help of make_poly_pointless, we would not have been able to use the induction hypotheses until much later in the proof, and the proof would have been dramatically longer. This also makes it more readable as you step through the proof, as we can seamlessly move between the original form including mulPP and the more functional form consisting of map and concat.

```
Lemma mulPP_comm : ∀ p q,
  mulPP p q = mulPP q p.
Proof.
  intros p q. repeat rewrite mulPP_mulPP''.
  generalize dependent q. induction p; induction q as [|m].
  - auto.
  - unfold mulPP'', mulMP'. simpl. rewrite (@concat_map_nil mono). auto.
  - unfold mulPP'', mulMP'. simpl. rewrite (@concat_map_nil mono). auto.
  - unfold mulPP''. simpl. rewrite (app_comm_cons _ _ (make_mono (a++m))).
    rewrite ← make_poly_pointless_r. rewrite mulPP''_refold. rewrite ← IHp.
    unfold mulPP''. rewrite make_poly_pointless_r. simpl. unfold mulMP' at 2.
    rewrite app_comm_cons. rewrite ← make_poly_pointless_r. rewrite mulPP''_refold.
```

```
    rewrite IHq. unfold mulPP''. rewrite make_poly_pointless_r. simpl.
    unfold mulMP' at 1. rewrite app_comm_cons. rewrite app_assoc.
    rewrite ← make_poly_pointless_r. rewrite mulPP''_refold. rewrite ← IHp.
    unfold mulPP''. rewrite make_poly_pointless_r. simpl. rewrite (app_assoc (map _
(map _ q))).
    apply Permutation_sort_eq. apply nodup_cancel_Permutation.
    apply Permutation_map. rewrite make_mono_app_comm. apply perm_skip.
    apply Permutation_app_tail. apply Permutation_app_comm.
Qed.
```

### 6.6.8 Axiom 8: Associativity of Multiplication

The eigth axiom states that, for all terms x y and z, $x \times (y \times z) = (x \times y) \times z$.

This one is also fairly complicated, so we will start small and build up to it. First, we prove a convenient side effect of make_poly_pointless, which allows us to simplify mulPP into a mulMP and a mulPP. Unlike commutativity, for this proof we opt to use the version of mulPP that includes a make_poly in its mulMP, in addition to the map make_mono version used previously.

```
Lemma mulPP''_cons : ∀ q a p,
  make_poly (mulMP' q a ++ mulPP'' q p) =
  mulPP'' q (a::p).
Proof.
  intros q a p. unfold mulPP''. rewrite make_poly_pointless_r. auto.
Qed.
```

Next is a deceptively easy lemma. map_app_make_poly is the primary application of nodup_cancel_map, proven in list_util. It states that if we are applying make_poly twice, we can remove the second application, even if there is a map app in between them. Clearly, here, the map app is in reference to mulMP.

```
Lemma map_app_make_poly : ∀ m p,
  (∀ a, In a p → is_mono a) →
  make_poly (map (app m) (make_poly p)) = make_poly (map (app m) p).
Proof.
  intros m p Hm. apply Permutation_sort_eq.
  apply Permutation_trans with (l':=(nodup_cancel mono_eq_dec (map make_mono
    (map (app m) (nodup_cancel mono_eq_dec (map make_mono p)))))).
    apply nodup_cancel_Permutation. repeat apply Permutation_map.
    unfold make_poly. rewrite ← Permutation_MonoSort_l. auto.
  rewrite (no_map_make_mono p); auto. repeat rewrite map_map. apply nodup_cancel_map.
Qed.
```

map_app_make_poly is then immediately applied here, to state that since mulMP'' already applies make_poly to its result, we can remove any make_poly calls inside.

```
Lemma mulMP''_make_poly : ∀ p m,
  (∀ a, In a p → is_mono a) →
  mulMP'' (make_poly p) m =
  mulMP'' p m.
Proof.
  intros p m. unfold mulMP''. apply map_app_make_poly.
Qed.
```

This very simple lemma states that since mulMP is effectively just a map, it distributes over append.

```
Lemma mulMP'_app : ∀ p q m,
  mulMP' (p ++ q) m =
  mulMP' p m ++ mulMP' q m.
Proof.
  intros p q m. unfold mulMP'. repeat rewrite map_app. auto.
Qed.
```

Now into the meat of the associativity proof. We begin by proving that mulMP' is associative. This proof is straightforward, and is proven by induction with the use of make_mono_pointless and Permutation_sort_mono_eq.

```
Lemma mulMP'_assoc : ∀ q a m,
  mulMP' (mulMP' q a) m =
  mulMP' (mulMP' q m) a.
Proof.
  intros q a m. unfold mulMP'. induction q.
  - auto.
  - simpl. repeat rewrite make_mono_pointless. f_equal.
    + apply Permutation_sort_mono_eq. apply Permutation_nodup.
      repeat rewrite app_assoc. apply Permutation_app_tail.
      apply Permutation_app_comm.
    + apply IHq.
Qed.
```

For the final associativity proof, we begin by using the commutativity lemma to make it so that $q$ is on the leftmost side of the multiplications. This means that it will never be the polynomial being mapped across, and allows us to do induction on just $p$ and $r$ instead of all three. $p$ becomes $a :: p$, and $r$ becomes $m :: r$.

The first three cases are easily solved with some rewrites and a call to auto, so we move on to the fourth. Similarly to the commutativity proof, the main struggle here is forcing mulPP to map across the same term on both sides of the equation. This is accomplished in a very similar way - by simplifying, using make_poly_pointless to get mulPP back in the goal, and then applying the two induction hypotheses to reorder the terms.

The crucial point is when we rewrite with mulMP'_mulMP'', allowing us to wrap our mulMPs in make_poly and make use of the lemmas we proved earlier in this section. This technique enables us to reorder the multiplications in a way that is convenient for us; $(q \times [a{::}p]) \times m$ becomes $(q \times a) \times m$ ++ $(q \times p) \times m$. At the end of all of this rewriting, we are left with the original $p \times q \times r$ as the last term of both sides, and $q \times p \times m$ and $q \times r \times a$ as the middle terms of both. These three terms are easily eliminated with the standard Permutation lemmas, because they are on both sides.

The only remaining challenge comes from the first term on each side; on the left, we have $(q \times a) \times m$, and on the right we have $(q \times m) \times a$. This is where the above mulMP'_assoc lemma comes into play, solving the last piece of the associativity lemma.

```
Lemma mulPP_assoc : ∀ p q r,
  mulPP (mulPP p q) r = mulPP p (mulPP q r).
Proof.
  intros p q r. rewrite (mulPP_comm _ (mulPP q _)). rewrite (mulPP_comm p _).
  generalize dependent r. induction p; induction r as [|m];
  repeat rewrite mulPP_0; repeat rewrite mulPP_0r; auto.
  repeat rewrite mulPP_mulPP'' in *. unfold mulPP''. simpl.
  repeat rewrite ← (make_poly_pointless_r _ (concat _)).
  repeat rewrite mulPP''_refold. repeat rewrite (mulPP''_cons q).
  pose (IHp (m::r)). repeat rewrite mulPP_mulPP'' in e. rewrite ← e.
  rewrite IHr. unfold mulPP'' at 2, mulPP'' at 4. simpl.
  repeat rewrite make_poly_pointless_r. repeat rewrite app_assoc.
  repeat rewrite ← (make_poly_pointless_r _ (concat _)).
  repeat rewrite mulPP''_refold. pose (IHp r). repeat rewrite mulPP_mulPP'' in e0.
  rewrite ← e0. repeat rewrite ← app_assoc. repeat rewrite mulMP'_mulMP''.
  repeat rewrite ← mulPP''_cons. repeat rewrite mulMP''_make_poly.
  repeat rewrite ← mulMP'_mulMP''. repeat rewrite app_assoc.
  apply Permutation_sort_eq. apply nodup_cancel_Permutation. apply Permutation_map.
  apply Permutation_app_tail. repeat rewrite mulMP'_app. rewrite mulMP'_assoc.
  repeat rewrite ← app_assoc. apply Permutation_app_head. apply Permutation_app_comm.
  intros a0 Hin. apply in_app_iff in Hin as []. unfold mulMP' in H.
  apply in_map_iff in H as [x[]]. rewrite ← H; auto.
  apply (make_poly_is_poly (concat (map (mulMP' q) r))). auto.
  intros a0 Hin. apply in_app_iff in Hin as []. unfold mulMP' in H.
  apply in_map_iff in H as [x[]]. rewrite ← H; auto.
  apply (make_poly_is_poly (concat (map (mulMP' q) p))). auto.
Qed.
```

### 6.6.9 Axiom 9: Multiplicative Identity - Self

Next comes the other multiplicative identity mentioned earlier. This axiom states that for all terms x, $x \times x = x$.

To begin, we prove that this holds for monomials; $m \times m = m$. This proof uses a combination of Permutation_Sorted_mono_eq and induction. We then use the standard Permutation lemmas to move the induction variable $a$ out to the front, and show that nodup removes one of the two $a$s. After that, perm_skip and the induction hypothesis solve the lemma.

```
Lemma make_mono_self : ∀ m,
  is_mono m →
  make_mono (m ++ m) = m.
Proof.
  intros m H. apply Permutation_Sorted_mono_eq.
  - induction m; auto. unfold make_mono. rewrite ← Permutation_VarSort_l. simpl.
    assert (In a (m++a::m)).
      intuition. destruct in_dec; try contradiction.
    apply Permutation_trans with (l':=(nodup var_eq_dec (a::m++m))).
       apply Permutation_nodup. apply Permutation_app_comm.
    simpl. assert (¬ In a (m++m)).
      apply NoDup_VarSorted in H as H1. apply NoDup_cons_iff in H1.
    intro. apply H1. apply in_app_iff in H2; intuition.
    destruct in_dec; try contradiction. apply perm_skip.
    apply Permutation_VarSort_l in IHm. auto. apply (mono_cons _ _ H).
  - apply VarSort.LocallySorted_sort.
  - apply Sorted_VarSorted. apply H.
Qed.
```

The full proof of the self multiplicative identity is much longer, but in a way very similar to the proof of commutativity. We begin by doing induction and simplifying, which distributes *one* of the induction variables across the list on the left side. This leaves us with $a \times a$ as the leftmost term, which is easily replaced with $a$ with the above lemma and then removed from both sides with perm_skip.

At this point we are left with a goal of the form $a*[a::p]$ ++ $[a::p]*p = p$ which is not particularly easy to deal with. However, by rewriting with mulPP_comm, we can force the second term on the left to simplify futher.

This leaves us with something along the lines of $a*[a::p]$ ++ $a*[a::p]$ ++ $p \times p = p$ which is much more workable! We know that $p \times p = p$ from the induction hypothesis, so this is then removed from both sides and all that is left is to prove that the same term added together twice is equal to an empty list. This follows from the nodup_cancel_self lemma used to prove addPP_p_p, and finished the proof of this lemma.

```
Lemma mulPP_p_p : ∀ p,
  is_poly p →
  mulPP p p = p.
Proof.
  intros p H. rewrite mulPP_mulPP'. rewrite mulPP'_mulPP''. apply Permutation_Sorted_eq.
  - induction p; auto. unfold mulPP'', make_poly. rewrite ← Permutation_MonoSort_l.
    simpl map at 1. apply poly_cons in H as H1. destruct H1. rewrite make_mono_self;
auto.
    rewrite no_make_mono; auto. rewrite map_app. apply Permutation_trans with
```

```
      (l':=(nodup_cancel mono_eq_dec (map make_mono (concat (map (mulMP' (a ::
      p)) p)) ++ a :: map make_mono (map make_mono (map (app a) p)))))).
      apply nodup_cancel_Permutation. rewrite app_comm_cons. apply Permutation_app_comm.
    rewrite ← nodup_cancel_pointless. apply Permutation_trans with (l':=(nodup_cancel
mono_eq_dec
      ((nodup_cancel mono_eq_dec (map make_mono (concat (map (mulMP' p) (a :: p)))))
      ++ (a :: map make_mono (map make_mono (map (app a) p)))))).
      apply nodup_cancel_Permutation. apply Permutation_app_tail. apply Permutation_sort_eq.
      repeat rewrite make_poly_refold. repeat rewrite mulPP''_refold.
      repeat rewrite ← mulPP'_mulPP''. repeat rewrite ← mulPP_mulPP'. apply mulPP_comm.
    rewrite nodup_cancel_pointless. apply Permutation_trans with (l':=
      (nodup_cancel mono_eq_dec (a :: map make_mono (map make_mono (map (app a)
p))
      ++ (map make_mono (concat (map (mulMP' p) (a :: p))))))).
      apply nodup_cancel_Permutation. apply Permutation_app_comm.
    simpl map. rewrite map_app. unfold mulMP' at 1. repeat rewrite (no_map_make_mono

    (map make_mono _)); try apply mono_in_map_make_mono. rewrite (app_assoc (map
_ _)).
    apply Permutation_trans with (l':=(nodup_cancel mono_eq_dec ((map make_mono (map

      (app a) p) ++ map make_mono (map (app a) p)) ++ a :: map make_mono (concat
      (map (mulMP' p) p))))). apply nodup_cancel_Permutation. apply Permutation_middle.
    rewrite ← nodup_cancel_pointless. rewrite nodup_cancel_self. simpl app.
    apply Permutation_trans with (l':=(nodup_cancel mono_eq_dec (map make_mono
      (concat (map (mulMP' p) p)) ++ [a]))). apply nodup_cancel_Permutation.
      replace (a::map make_mono (concat (map (mulMP' p) p))) with ([a] ++ map
      make_mono (concat (map (mulMP' p) p))); auto. apply Permutation_app_comm.
    rewrite ← nodup_cancel_pointless. apply Permutation_trans with (l':=(nodup_cancel
      mono_eq_dec (p ++ [a]))). apply nodup_cancel_Permutation.
      apply Permutation_app_tail. unfold mulPP'', make_poly in IHp.
      rewrite ← Permutation_MonoSort_l in IHp. apply IHp; auto.
    replace (a::p) with ([a]++p); auto. rewrite no_nodup_cancel_NoDup.
    apply Permutation_app_comm. apply Permutation_NoDup with (l:=(a::p)).
    replace (a::p) with ([a]++p); auto. apply Permutation_app_comm.
    destruct H. apply NoDup_MonoSorted in H. auto.
  - unfold make_poly. apply LocallySorted_sort.
  - apply Sorted_MonoSorted. apply H.
Qed.
```

### 6.6.10 Axiom 10: Distribution

Finally, we are left with the most intimidating of the axioms - distribution. This states, as one would expect, that for all terms x y and z, $x \times (y + z) = (x \times y) + (x \times z)$.

In a similar approach to what we have done for some of the other lemmas, we begin by proving this on a smaller scale, working with just mulMP and addPP. This lemma is once again solved easily by the map_app_make_poly we proved while working on multiplication associativity, combined with make_poly_pointless.

```
Lemma mulMP''_distr_addPP : ∀ m p q,
  is_poly p → is_poly q →
  mulMP'' (addPP p q) m = addPP (mulMP'' p m) (mulMP'' q m).
Proof.
  intros m p q Hp Hq. unfold mulMP'', addPP. rewrite map_app_make_poly.
  rewrite make_poly_pointless. rewrite make_poly_app_comm.
  rewrite make_poly_pointless. rewrite make_poly_app_comm.
  rewrite map_app. auto. intros a Hin. apply in_app_iff in Hin as [].
  apply Hp. auto. apply Hq. auto.
Qed.
```

For the distribution proof itself, we begin by performing induction on $r$, the element outside of the addPP call initially. We begin by simplifying, and using the usual combination of make_poly_pointless and refolding to convert our goal to a form of $(p{+}q)^{*}a$ ++ $(p{+}q)^{*}r$ .

We then apply similar tactics on the right side, to convert our goal to a form similar to $(p{\times}a) + (q{\times}a) + (p{\times}r) + (q{\times}r)$ . The two terms containing $r$ are easy to deal with, since we know they are equal to the $(p{+}q)^{*}r$ we have on the left side due to the induction hypothesis. Similarly, the first two terms are known to be equal to $(p{+}q)^{*}a$ from the *mulMP_distr_addPP* lemma we just proved. This results in us having the same thing on both sides, thus solving the final of the ten B-Unification axioms.

```
Lemma mulPP_distr_addPP : ∀ p q r,
  is_poly p → is_poly q →
  mulPP (addPP p q) r = addPP (mulPP p r) (mulPP q r).
Proof.
  intros p q r Hp Hq. induction r; auto. rewrite mulPP_mulPP''. unfold mulPP''.
  simpl. rewrite mulPP_mulPP'', (mulPP_mulPP'' q), make_poly_app_comm.
  rewrite ← make_poly_pointless. rewrite make_poly_app_comm.
  rewrite mulPP''_refold.
  rewrite addPP_refold. repeat unfold mulPP'' at 2. simpl. unfold addPP at 4.
  rewrite make_poly_pointless. rewrite addPP_refold.
  rewrite (addPP_comm _ (make_poly _)).
  unfold addPP at 4. rewrite make_poly_pointless. rewrite ← app_assoc.
  rewrite make_poly_app_comm. rewrite ← app_assoc.
  rewrite ← make_poly_pointless.
  rewrite mulPP''_refold. rewrite ← app_assoc. rewrite app_assoc.
```

```
  rewrite make_poly_app_comm.
  rewrite ← app_assoc. rewrite ← make_poly_pointless. rewrite mulPP''_refold.
  replace (make_poly (mulPP'' p r ++ mulMP' q a ++ mulPP'' q r ++ mulMP' p a))
    with (make_poly ((mulPP'' p r ++ mulPP'' q r) ++ mulMP' p a ++ mulMP' q a)).
  rewrite ← make_poly_pointless. rewrite (addPP_refold (mulPP'' _ _)).
  rewrite make_poly_app_comm. rewrite addPP_refold.
  rewrite mulPP_mulPP'', (mulPP_mulPP'' p), (mulPP_mulPP'' q) in IHr.
  rewrite ← IHr. unfold addPP at 4.
  rewrite ← make_poly_pointless. unfold addPP. repeat rewrite mulMP'_mulMP''.
  rewrite (make_poly_app_comm (mulMP'' _ _) (mulMP' _ _)).
  rewrite mulMP'_mulMP''. rewrite (make_poly_app_comm (mulMP'' _ _) (mulMP'' _ _)).
  repeat rewrite addPP_refold. f_equal. apply mulMP''_distr_addPP; auto.
  apply make_poly_Permutation. rewrite ← app_assoc.
  apply Permutation_app_head. rewrite app_assoc.
  apply Permutation_trans with
    (l':=mulMP' q a ++ mulPP'' q r ++ mulMP' p a).
  apply Permutation_app_comm.
  auto.
Qed.
```

For convenience, we also prove that distribution can be applied right, which follows from mulPP_comm and the distribution lemma we just proved.

```
Lemma mulPP_distr_addPPr : ∀ p q r,
  is_poly p → is_poly q →
  mulPP r (addPP p q) = addPP (mulPP r p) (mulPP r q).
Proof.
  intros p q r Hp Hq. rewrite mulPP_comm. rewrite (mulPP_comm r p).
  rewrite (mulPP_comm r q). apply mulPP_distr_addPP; auto.
Qed.
```

## 6.7 Other Facts About Arithmetic

```
Lemma mulPP_mono_cons : ∀ x m,
  is_mono (x :: m) →
  mulPP [[x]] [m] = [x :: m].
Proof.
  intros x m H. unfold mulPP, distribute. simpl. apply Permutation_Sorted_eq.
  - apply Permutation_trans with (l':=(nodup_cancel mono_eq_dec (map make_mono [m++[x]]))).
    apply Permutation_sym. apply Permuted_sort. rewrite no_nodup_cancel_NoDup.
    simpl. assert (make_mono (m++[x]) = x::m).
    + rewrite ← no_make_mono; auto. apply Permutation_sort_mono_eq.
      repeat rewrite no_nodup_NoDup. replace (x::m) with ([x]++m); auto; apply
```

```
Permutation_app_comm.
      apply NoDup_VarSorted; apply H. apply Permutation_NoDup with (l:=(x::m)).
      replace (x::m) with ([x]++m); auto; apply Permutation_app_comm.
      apply NoDup_VarSorted; apply H.
    + rewrite H0. auto.
    + apply NoDup_cons; auto.
  - apply LocallySorted_sort.
  - apply Sorted_cons; auto.
Qed.

Lemma addPP_poly_cons : ∀ m p,
  is_poly (m :: p) →
  addPP [m] p = m :: p.
Proof.
  intros m p H. unfold addPP. simpl. rewrite no_make_poly; auto.
Qed.

Hint Resolve addPP_is_poly mulPP_is_poly.

Lemma mulPP_addPP_1 : ∀ p q r,
  is_poly p → is_poly q → is_poly r →
  mulPP (addPP (mulPP p q) r) (addPP [[]] q) =
  mulPP (addPP [[]] q) r.
Proof.
  intros p q r Hp Hq Hr. rewrite mulPP_distr_addPP; auto.
  rewrite mulPP_distr_addPPr; auto. rewrite mulPP_1r; auto.
  rewrite mulPP_assoc. rewrite mulPP_p_p; auto. rewrite addPP_p_p; auto.
  rewrite addPP_0; auto. rewrite mulPP_comm. auto.
Qed.

Lemma make_poly_rem_vars : ∀ p x,
  In x (vars (make_poly p)) →
  In x (vars p).
Proof.
  intros p x H. induction p.
  - inversion H.
  - unfold vars. simpl. apply nodup_In. apply in_app_iff.
    unfold vars, make_poly in H. apply nodup_In in H.
    apply In_concat_exists in H as [m []].
    apply In_sorted in H. apply nodup_cancel_in in H.
    apply in_map_iff in H as [n []]. destruct H1.
    + left. apply make_mono_In. rewrite H1. rewrite H. auto.
    + right. apply In_concat_exists. ∃ n. split; auto. apply make_mono_In.
      rewrite H. auto.
Qed.
```

```
Lemma incl_vars_addPP : ∀ p q xs,
  incl (vars p) xs ∧ incl (vars q) xs →
  incl (vars (addPP p q)) xs.
Proof.
  unfold incl, addPP.
  intros p q xs [HinP HinQ] x HinPQ.
  apply make_poly_rem_vars in HinPQ.
  unfold vars in HinPQ.
  apply nodup_In in HinPQ.
  rewrite concat_app in HinPQ.
  apply in_app_or in HinPQ as [Hin | Hin].
  - apply HinP. apply nodup_In. auto.
  - apply HinQ. apply nodup_In. auto.
Qed.

Lemma In_distribute : ∀ (l m:poly) a,
  In a (vars (distribute l m)) →
  In a (vars l) ∨ In a (vars m).
Proof.
  intros l m a H. unfold distribute, vars in H. apply nodup_In in H.
  apply In_concat_exists in H. destruct H as [ll[]].
  apply In_concat_exists in H. destruct H as [ll1[]].
  apply in_map_iff in H. destruct H as [x[]]. rewrite ← H in H1.
  apply in_map_iff in H1. destruct H1 as [x0[]]. rewrite ← H1 in H0.
  apply in_app_iff in H0. destruct H0.
  - right. apply nodup_In. apply In_concat_exists. ∃ x. auto.
  - left. apply nodup_In. apply In_concat_exists. ∃ x0. auto.
Qed.

Lemma incl_vars_mulPP : ∀ p q xs,
  incl (vars p) xs ∧ incl (vars q) xs →
  incl (vars (mulPP p q)) xs.
Proof.
  unfold incl, mulPP.
  intros p q xs [HinP HinQ] x HinPQ.
  apply make_poly_rem_vars in HinPQ.
  apply In_distribute in HinPQ. destruct HinPQ.
  - apply HinP. auto.
  - apply HinQ. auto.
Qed.

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

```
Proof.
  intros f p l r H H0. apply Permutation_Sorted_eq.
  - generalize dependent l; generalize dependent r. induction p; intros.
    + simpl in H0. inversion H0. auto.
    + assert (H1:=H0); auto. apply partition_Permutation in H1. simpl in H0.
      destruct (partition f p) as [g d]. unfold addPP, make_poly.
      rewrite <- Permutation_MonoSort_r. rewrite unsorted_poly. destruct (f a); inversion
H0.
      * rewrite <- H3 in H1. apply H1.
      * rewrite <- H4 in H1. apply H1.
      * destruct H. apply NoDup_MonoSorted in H. apply (Permutation_NoDup H1 H).
      * intros m Hin. apply H. apply Permutation_sym in H1. apply (Permutation_in _
H1 Hin).
  - apply Sorted_MonoSorted. apply H.
  - apply Sorted_MonoSorted. apply make_poly_is_poly.
Qed.

Lemma part_is_poly : forall f p l r,
  is_poly p ->
  partition f p = (l, r) ->
  is_poly l /\ is_poly r.
Proof.
  intros f p l r Hpoly Hpart. destruct Hpoly. split; split.
  - apply (part_Sorted _ _ _ mono_lt_Transitive H _ _ Hpart).
  - intros m Hin. apply H0. apply elements_in_partition with (x:=m) in Hpart.
    apply Hpart; auto.
  - apply (part_Sorted _ _ _ mono_lt_Transitive H _ _ Hpart).
  - intros m Hin. apply H0. apply elements_in_partition with (x:=m) in Hpart.
    apply Hpart; auto.
Qed.

Lemma remove_Sorted_eq : forall x (l l':mono),
  is_mono l -> is_mono l' ->
  In x l <-> In x l' ->
  remove var_eq_dec x l = remove var_eq_dec x l' ->
  l = l'.
Proof.
  intros x l l' Hl Hl' Hx Hrem.
  generalize dependent l'; induction l; induction l'; intros.
  - auto.
  - destruct (var_eq_dec x a) eqn:Heq.
    + rewrite e in Hx. exfalso. apply Hx. intuition.
    + simpl in Hrem. rewrite Heq in Hrem. inversion Hrem.
  - destruct (var_eq_dec x a) eqn:Heq.
```

```
    + rewrite e in Hx. exfalso. apply Hx. intuition.
    + simpl in Hrem. rewrite Heq in Hrem. inversion Hrem.
  - clear IHl'. destruct (var_eq_dec a a0).
    + rewrite e. f_equal. rewrite e in Hrem. simpl in Hrem.
      apply mono_cons in Hl as Hl1. apply mono_cons in Hl' as Hl'1.
      destruct (var_eq_dec x a0).
      × apply IHl; auto. apply NoDup_VarSorted in Hl. apply NoDup_cons_iff in Hl.
        rewrite e in Hl. rewrite ← e0 in Hl. destruct Hl. split; intro. contradiction.
        apply NoDup_VarSorted in Hl'. apply NoDup_cons_iff in Hl'.
        rewrite ← e0 in Hl'. destruct Hl'. contradiction.
      × inversion Hrem. apply IHl; auto. destruct Hx. split; intro. simpl in H.
        rewrite e in H. destruct H; auto. rewrite H in n. contradiction.
        simpl in H1. rewrite e in H1. destruct H1; auto. rewrite H1 in n.
contradiction.
    + destruct (in_dec var_eq_dec x (a::l)).
      × apply Hx in i as i'. apply in_split in i. apply in_split in i'.
        destruct i as [l1[l2 i]]. destruct i' as [l1'[l2' i']].
        pose (NoDup_VarSorted _ Hl). pose (NoDup_VarSorted _ Hl').
        apply (NoDup_In_split _ _ _ _ i) in n0 as []. apply (NoDup_In_split _ _ _ _ i') in
n1 as [].
        rewrite i in Hrem. rewrite i' in Hrem. repeat rewrite remove_distr_app in
Hrem.
        simpl in Hrem. destruct (var_eq_dec x x); try contradiction.
        rewrite not_In_remove in Hrem; auto. rewrite not_In_remove in Hrem; auto.
        rewrite not_In_remove in Hrem; auto. rewrite not_In_remove in Hrem; auto.
        destruct l1; destruct l1'; simpl in i; simpl in i'; simpl in Hrem;
        inversion i; inversion i'.
        – rewrite H4 in n. rewrite H6 in n. contradiction.
        – rewrite H7 in Hl'. rewrite i in Hl. rewrite Hrem in Hl.
          rewrite H6 in Hl'. assert (x < v). apply Sorted_inv in Hl as [].
          apply HdRel_inv in H8. auto. assert (v < x). apply Sorted_StronglySorted in
Hl'.
          apply StronglySorted_inv in Hl' as []. rewrite Forall_forall in H9.
          apply H9. intuition. apply lt_Transitive. apply lt_asymm in H8. contradiction.
        – rewrite H7 in Hl'. rewrite i in Hl. rewrite ← Hrem in Hl'.
          rewrite H6 in Hl'. assert (n0 < x). apply Sorted_StronglySorted in Hl.
          apply StronglySorted_inv in Hl as []. rewrite Forall_forall in H8.
          apply H8. intuition. apply lt_Transitive. assert (x < n0).
          apply Sorted_inv in Hl' as []. apply HdRel_inv in H9; auto.
          apply lt_asymm in H8. contradiction.
        – inversion Hrem. rewrite ← H4 in H8. rewrite ← H6 in H8. contradiction.
      × assert (¬In x (a0::l')). intro. apply n0. apply Hx. auto.
```

```
      rewrite not_In_remove in Hrem; auto. rewrite not_In_remove in Hrem; auto.
Qed.

Lemma NoDup_map_remove : ∀ x p,
  is_poly p →
  (∀ m, In m p → In x m) →
  NoDup (map (remove var_eq_dec x) p).
Proof.
  intros x p Hp Hx. induction p.
  - simpl. auto.
  - simpl. apply NoDup_cons.
    + intro. apply in_map_iff in H. destruct H as [y []]. assert (y = a).
      × apply poly_cons in Hp. destruct Hp. unfold is_poly in H1. destruct H1.
        apply H3 in H0 as H4. apply (remove_Sorted_eq x); auto. split; intro.
        apply Hx. intuition. apply Hx. intuition.
      × rewrite H1 in H0. unfold is_poly in Hp. destruct Hp.
        apply NoDup_MonoSorted in H2 as H4. apply NoDup_cons_iff in H4 as [].
        contradiction.
    + apply IHp.
      × apply poly_cons in Hp. apply Hp.
      × intros m H. apply Hx. intuition.
Qed.

Lemma NoDup_map_app : ∀ x l,
  is_poly l →
  (∀ m, In m l → ¬ In x m) →
  NoDup (map make_mono (map (fun a : list var ⇒ a ++ [x]) l)).
Proof.
  intros x l Hp Hin. induction l.
  - simpl. auto.
  - simpl. apply NoDup_cons.
    + intros H. rewrite map_map in H. apply in_map_iff in H as [m []]. assert (a=m).
      × apply poly_cons in Hp as []. apply Permutation_Sorted_mono_eq.
        – apply Permutation_sort_mono_eq in H. rewrite no_nodup_NoDup in H.
          rewrite no_nodup_NoDup in H.
          ++ pose (Permutation_cons_append m x). pose (Permutation_cons_append a
x).
             apply (Permutation_trans p) in H. apply Permutation_sym in p0.
             apply (Permutation_trans H) in p0. apply Permutation_cons_inv in p0.
             apply Permutation_sym. auto.
          ++ apply Permutation_NoDup with (l:=(x::a)). apply Permutation_cons_append.
             apply NoDup_cons. apply Hin. intuition. unfold is_mono in H2.
             apply NoDup_VarSorted in H2. auto.
          ++ apply Permutation_NoDup with (l:=(x::m)). apply Permutation_cons_append.
```

apply NoDup_cons. apply *Hin*. intuition. unfold is_poly in *H1*.
destruct *H1*. apply *H3* in *H0*. unfold is_mono in *H0*.
apply NoDup_VarSorted in *H0*. auto.
– unfold is_mono in *H2*. apply Sorted_VarSorted. auto.
– unfold is_poly in *H1*. destruct *H1*. apply *H3* in *H0*. apply Sorted_VarSorted.
auto.
× rewrite ← *H1* in *H0*. unfold is_poly in *Hp*. destruct *Hp*.
apply NoDup_MonoSorted in *H2*. apply NoDup_cons_iff in *H2* as []. *contradiction*.
+ apply *IHl*. apply poly_cons in *Hp*. apply *Hp*. intros *m H*. apply *Hin*. intuition.
Qed.

Lemma mulPP_Permutation : ∀ *x a0 l*,
is_poly (*a0* : : *l*) →
(∀ *m*, In *m* (*a0* : : *l*) → ¬ In *x m*) →
**Permutation** (mulPP [[*x*]] (*a0* : : *l*)) ((make_mono (*a0*++[*x*])) : : (mulPP [[*x*]] *l*)).
Proof.
intros *x a0 l Hp Hx*. unfold mulPP, distribute. simpl. unfold make_poly.
pose (MonoSort.Permuted_sort (nodup_cancel mono_eq_dec
(map make_mono ((*a0* ++ [*x*]) : : concat (map (fun *a* : **list** var ⇒ [*a* ++ [*x*]])
*l*))))).
apply Permutation_sym in *p*. apply (Permutation_trans *p*). simpl *map*.
rewrite no_nodup_cancel_NoDup; clear *p*.
- apply perm_skip. apply Permutation_trans with (*l'*:=(nodup_cancel mono_eq_dec (map
make_mono (concat (map (fun *a* : **list** var ⇒ [*a* ++ [*x*]]) *l*))))).
+ rewrite no_nodup_cancel_NoDup; auto. rewrite concat_map. apply NoDup_map_app.
apply poly_cons in *Hp*. apply *Hp*. intros *m H*. apply *Hx*. intuition.
+ apply MonoSort.Permuted_sort.
- rewrite ← map_cons. rewrite concat_map.
rewrite ← map_cons with (*f*:=(fun *a* : **list** var ⇒ *a* ++ [*x*])).
apply NoDup_map_app; auto.
Qed.

Lemma mulPP_map_app_permutation : ∀ (*x*:var) (*l l'* : poly),
is_poly *l* →
(∀ *m*, In *m l* → ¬ In *x m*) →
**Permutation** *l l'* →
**Permutation** (mulPP [[*x*]] *l*) (map (fun *a* ⇒ (make_mono (*a* ++ [*x*]))) *l'*).
Proof.
intros *x l l' Hp H H0*. generalize dependent *l'*. induction *l*; induction *l'*.
- intros. unfold mulPP, distribute, make_poly, MonoSort.sort. simpl. auto.
- intros. apply Permutation_nil_cons in *H0*. *contradiction*.
- intros. apply Permutation_sym in *H0*. apply Permutation_nil_cons in *H0*. *contradiction*.
- intros. clear *IHl'*. destruct (mono_eq_dec *a a0*).
+ rewrite *e* in *. pose (mulPP_Permutation *x a0 l Hp H*). apply (Permutation_trans

```
p). simpl.
      apply perm_skip. apply IHl.
      × clear p. apply poly_cons in Hp. apply Hp.
      × intros m Hin. apply H. intuition.
      × apply Permutation_cons_inv in H0. auto.
    + apply Permutation_incl in H0 as H1. destruct H1. apply incl_cons_inv in H1 as
[].
      destruct H1; try (rewrite H1 in n; contradiction). apply in_split in H1.
      destruct H1 as [l1 [l2]]. rewrite H1 in H0.
      pose (Permutation_middle (a0::l1) l2 a). apply Permutation_sym in p.
      simpl in p. apply (Permutation_trans H0) in p.
      apply Permutation_cons_inv in p. rewrite H1. simpl. rewrite map_app. simpl.
      pose (Permutation_middle ((make_mono (a0 ++ [x]) :: map
        (fun a1 : list var ⇒ make_mono (a1 ++ [x])) l1)) (map
        (fun a1 : list var ⇒ make_mono (a1 ++ [x])) l2) (make_mono (a++[x]))).
      simpl in p0. simpl. apply Permutation_trans with (l':=(make_mono (a ++ [x])
      :: make_mono (a0 ++ [x])
          :: map (fun a1 : list var ⇒ make_mono (a1 ++ [x])) l1 ++
             map (fun a1 : list var ⇒ make_mono (a1 ++ [x])) l2)); auto. clear p0.
      rewrite ← map_app. rewrite ← (map_cons (fun a1 : list var ⇒ make_mono (a1
++ [x])) a0 (@app (list var) l1 l2)).
      pose (mulPP_Permutation x a l Hp H). apply (Permutation_trans p0). apply perm_skip.
      apply IHl.
      × clear p0. apply poly_cons in Hp. apply Hp.
      × intros m Hin. apply H. intuition.
      × apply p.
Qed.

Lemma map_app_remove_Permutation : ∀ p x,
  is_poly p →
  (∀ m, In m p → In x m) →
  Permutation p (map (fun a ⇒ (make_mono (a ++ [x]))) (map (remove var_eq_dec x)
p)).
Proof.
  intros p x H H0. rewrite map_map. induction p.
  - auto.
  - simpl. assert (make_mono (@app var (remove var_eq_dec x a) [x]) = a).
    + unfold make_mono. rewrite no_nodup_NoDup.
      × apply Permutation_Sorted_mono_eq.
        – apply Permutation_trans with (l':=(remove var_eq_dec x a ++ [x])).
          apply Permutation_sym. apply VarSort.Permuted_sort.
          pose (in_split x a). destruct e as [l1 [l2 e]]. apply H0. intuition.
          rewrite e. apply Permutation_trans with (l':=(x::remove var_eq_dec x (l1++x::l2))).
```

```
          apply Permutation_sym. apply Permutation_cons_append.
          apply Permutation_trans with (l':=(x::l1++l2)). apply perm_skip.
          rewrite remove_distr_app. replace (x::l2) with ([x]++l2); auto.
          rewrite remove_distr_app. simpl. destruct (var_eq_dec x x); try contradiction.
          rewrite app_nil_l. repeat rewrite not_In_remove; try apply Permutation_refl;
          try (apply poly_cons in H as []; unfold is_mono in H1;
          apply NoDup_VarSorted in H1; rewrite e in H1; apply NoDup_remove_2 in
H1).
          intros x2. apply H1. intuition. intros x1. apply H1. intuition.
          apply Permutation_middle.
        - apply VarSort.LocallySorted_sort.
        - apply poly_cons in H as []. unfold is_mono in H1.
          apply Sorted_VarSorted. auto.
      × apply Permutation_NoDup with (l:=(x::remove var_eq_dec x a)).
        apply Permutation_cons_append. apply NoDup_cons.
        apply remove_In. apply NoDup_remove. apply poly_cons in H as [].
        unfold is_mono in H1. apply NoDup_VarSorted. auto.
    + rewrite H1. apply perm_skip. apply IHp.
      × apply poly_cons in H. apply H.
      × intros m Hin. apply H0. intuition.
Qed.
```

# Chapter 7

# Library B_Unification.poly_unif

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

```
Require Export poly.
```

## 7.1 Introduction

This section deals with defining substitutions and their properties using a polynomial representation. As with the inductive term representation, substitutions are just list of replacements, where variables are swapped with polynomials instead of terms. Crucial to the proof of correctness in the following chapter, substitution is proven to distribute over polynomial addition and multiplication. Definitions are provided for unifier, unifiable, and properties relating multiple substitutions such as more general and composition.

## 7.2 Substitution Definitions

A *substitution* is defined as a list of replacements. A *replacement* is just a tuple of a variable and a polynomial.

```
Definition repl := prod var poly.
```

```
Definition subst := list repl.
```

Since the poly data type doesn't enforce the properties of actual polynomials, the is_poly predicate is used to check if a term is in polynomial form. Likewise, the is_poly_subst predicate below verifies that every term in the range of the substitution is a polynomial.

```
Definition is_poly_subst (s : subst) : Prop :=
  ∀ x p, In (x, p) s → is_poly p.
```

The next three functions implement how substitutions are applied to terms. At the top level, substP applies a substitution to a polynomial by calling substM on each monomial. From there, substV is called on each variable. Because variables and monomials are converted to polynomials, the process isn't simplying mapping application across the lists. substM and substP must multiply and add each polynomial together respectively.

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

Fixpoint substM (s : subst) (m : mono) : poly :=
  match m with
  | [] ⇒ [[]]
   | x :: m ⇒ mulPP (substV s x) (substM s m)
  end.

Definition substP (s : subst) (p : poly) : poly :=
  make_poly (concat (map (substM s) p)).
```

Useful in later proofs is the ability to rewrite the unfolded definition of substP as just the function call.

```
Lemma substP_refold : ∀ s p,
  make_poly (concat (map (substM s) p)) = substP s p.
Proof. auto. Qed.
```

The following lemmas state that substitution applications always produce polynomials. This fact is necessary for proving distribution and other properties of substitutions.

```
Lemma substV_is_poly : ∀ x s,
  is_poly_subst s →
  is_poly (substV s x).
Proof.
  intros x s H. unfold is_poly_subst in H. induction s; simpl; auto.
  destruct a eqn:Ha. destruct (x =? v).
  - apply (H v). intuition.
  - apply IHs. intros x0 p0 H0. apply (H x0). intuition.
Qed.

Lemma substM_is_poly : ∀ s m,
  is_poly (substM s m).
Proof.
  intros s m. unfold substM; destruct m; auto.
Qed.

Lemma substP_is_poly : ∀ s p,
  is_poly (substP s p).
```

```
Proof.
  intros. unfold substP. auto.
Qed.

Hint Resolve substP_is_poly substM_is_poly.
```

The lemma below states that a substitution applied to a variable in polynomial form is equivalent to the substitution applied to just the variable. This fact only holds when the substitution's range consists of polynomials.

```
Lemma subst_var_eq : ∀ x s,
  is_poly_subst s →
  substP s [[x]] = substV s x.
Proof.
  intros. simpl.
  apply (substV_is_poly x s) in H. unfold substP. simpl. rewrite app_nil_r.
  rewrite mulPP_1r; auto. rewrite no_make_poly; auto.
Qed.
```

The next two lemmas deal with simplifying substitutions where the first replacement tuple is useless for the given term. This is the case when the variable being replaced is not present in the term. It allows the replacement to be dropped from the substitution without changing the result.

```
Lemma substM_cons : ∀ x m,
  ¬ In x m →
  ∀ p s, substM ((x, p) :: s) m = substM s m.
Proof.
  intros. induction m; auto. simpl. f_equal.
  - destruct (a =? x) eqn:H0; auto.
    symmetry in H0. apply beq_nat_eq in H0. exfalso.
    simpl in H. apply H. left. auto.
  - apply IHm. intro. apply H. right. auto.
Qed.

Lemma substP_cons : ∀ x p,
  (∀ m, In m p → ¬ In x m) →
  ∀ q s, substP ((x, q) :: s) p = substP s p.
Proof.
  intros. induction p; auto. unfold substP. simpl.
  repeat rewrite ← (make_poly_pointless_r _ (concat _)). f_equal. f_equal.
  - apply substM_cons. apply H. left. auto.
  - apply IHp. intros. apply H. right. auto.
Qed.
```

Substitutions applied to constants have no effect.

```
Lemma substP_1 : ∀ s,
```

```
  substP s [[]] = [[]].
Proof.
  intros. unfold substP. simpl. auto.
Qed.

Lemma substP_0 : ∀ s,
  substP s [] = [].
Proof.
  intros. unfold substP. simpl. auto.
Qed.
```

The identity substitution–the empty list–has no effect when applied to a term.

```
Lemma empty_substM : ∀ m,
  is_mono m →
  substM [] m = [m].
Proof.
  intros. induction m; auto. simpl.
  apply mono_cons in H as H0.
  rewrite IHm; auto.
  apply mulPP_mono_cons; auto.
Qed.

Lemma empty_substP : ∀ p,
  is_poly p →
  substP [] p = p.
Proof.
  intros. induction p; auto. unfold substP. simpl.
  apply poly_cons in H as H0. destruct H0.
  rewrite ← make_poly_pointless_r. rewrite substP_refold.
  rewrite IHp; auto. rewrite empty_substM; auto.
  apply addPP_poly_cons; auto.
Qed.
```

## 7.3 Distribution Over Arithmetic Operators

Below is the statement and proof that substitution distributes over polynomial addition. Given a substitution $s$ and two terms in polynomial form $p$ and $q$, it is shown that $s(p+q)\downarrow_P = (s(p)+s(q))\downarrow_P$. The proof relies heavily on facts about permutations proven in the list_util library.

```
Lemma substP_distr_addPP : ∀ p q s,
  is_poly p →
  is_poly q →
  substP s (addPP p q) = addPP (substP s p) (substP s q).
Proof.
```

```
  intros p q s Hp Hq. unfold substP, addPP.
  apply Permutation_sort_eq. apply Permutation_trans with (l':=
    (nodup_cancel mono_eq_dec (map make_mono (concat (map (substM s)
    (nodup_cancel mono_eq_dec (map make_mono (p ++ q)))))))).
    apply nodup_cancel_Permutation. apply Permutation_map.
    apply Permutation_concat. apply Permutation_map. unfold make_poly.
    rewrite ← Permutation_MonoSort_l. auto.
  apply Permutation_sym. apply Permutation_trans with (l':=(nodup_cancel
    mono_eq_dec (map make_mono (nodup_cancel mono_eq_dec (map make_mono (concat
    (map (substM s) (p)))) ++ (nodup_cancel mono_eq_dec (map make_mono (concat
    (map (substM s) q)))))))). apply nodup_cancel_Permutation.
    apply Permutation_map. apply Permutation_app; unfold make_poly;
    rewrite ← Permutation_MonoSort_l; auto.
  rewrite (no_map_make_mono ((nodup_cancel _ _) ++ (nodup_cancel _ _))).
  rewrite nodup_cancel_pointless. apply Permutation_trans with (l':=
    (nodup_cancel mono_eq_dec (nodup_cancel mono_eq_dec (map make_mono (concat
    (map (substM s) q))) ++ map make_mono (concat (map (substM s) p))))).
    apply nodup_cancel_Permutation. apply Permutation_app_comm.
  rewrite nodup_cancel_pointless. rewrite ← map_app. rewrite ← concat_app.
  rewrite ← map_app. rewrite (no_map_make_mono (p++q)).
  apply Permutation_trans with (l':=(nodup_cancel mono_eq_dec (map make_mono
    (concat (map (substM s) (p ++ q)))))). apply nodup_cancel_Permutation.
    apply Permutation_map. apply Permutation_concat. apply Permutation_map.
    apply Permutation_app_comm.
  apply Permutation_sym. repeat rewrite List.concat_map.
  repeat rewrite map_map. apply nodup_cancel_concat_map.
  intros x. rewrite no_map_make_mono. apply NoDup_MonoSorted;
    apply substM_is_poly.
  intros m Hin. apply (substM_is_poly s x); auto.
  intros m Hin. apply in_app_iff in Hin as []; destruct Hp; destruct Hq; auto.
  intros m Hin. apply in_app_iff in Hin as []; apply nodup_cancel_in in H;
    apply mono_in_map_make_mono in H; auto.
Qed.
```

The next six lemmas deal with proving that substitution distributes over polynomial multiplication. Given a substitution $s$ and two terms in polynomial form $p$ and $q$, it is shown that $s(p*q)\downarrow_P = (s(p)*s(q))\downarrow_P$. The proof turns out to be much more difficult than the one for addition because the underlying arithmetic operation is more complex.

If two monomials are permutations (obviously not in monomial form), then applying any substitution to either will produce the same result. A weaker form that follows from this is that the results are permutations as well.

```
Lemma substM_Permutation_eq : ∀ s m n,
  Permutation m n →
```

```
  substM s m = substM s n.
Proof.
  intros s m n H. induction H; auto.
  - simpl. rewrite IHPermutation. auto.
  - simpl. rewrite mulPP_comm. rewrite mulPP_assoc.
    rewrite (mulPP_comm (substM s l)). auto.
  - rewrite IHPermutation1. rewrite IHPermutation2. auto.
Qed.

Lemma substM_Permutation : ∀ s m n,
  Permutation m n →
  Permutation (substM s m) (substM s n).
Proof.
  intros s m n H. rewrite (substM_Permutation_eq s m n); auto.
Qed.
```

Adding duplicate variables to a monomial doesn't change the result of applying a substitution. This is only true if the substitution's range only has polynomials.

```
Lemma substM_nodup_pointless : ∀ s m,
  is_poly_subst s →
  substM s (nodup var_eq_dec m) = substM s m.
Proof.
  intros s m Hps. induction m; auto. simpl. destruct in_dec.
  - apply in_split in i. destruct i as [l1 [l2 H]].
    assert (Permutation m (a :: l1 ++ l2)). rewrite H. apply Permutation_sym.
      apply Permutation_middle.
    apply substM_Permutation_eq with (s:=s) in H0. rewrite H0. simpl.
    rewrite (mulPP_comm _ (substM _ _)). rewrite mulPP_comm.
    rewrite mulPP_assoc. rewrite mulPP_p_p. rewrite mulPP_comm. rewrite IHm.
    rewrite H0. simpl. auto. apply substV_is_poly. auto.
  - simpl. rewrite IHm. auto.
Qed.
```

The idea behind the following two lemmas is that substitutions distribute over multiplication of a monomial and polynomial. The specifics of both are convoluted yet easier to prove than distribution over two polynomials.

```
Lemma substM_distr_mulMP : ∀ m n s,
  is_poly_subst s →
  is_mono n →
  Permutation
    (nodup_cancel mono_eq_dec (map make_mono (substM s (make_mono
      (make_mono (m ++ n))))))
    (nodup_cancel mono_eq_dec (map make_mono (concat (map (mulMP''
      (map make_mono (substM s m))) (map make_mono (substM s n)))))).
```

```
Proof.
  intros m n s Hps H. rewrite (no_make_mono (make_mono (m ++ n))); auto.
  repeat rewrite (no_map_make_mono (substM s _)); auto. apply Permutation_trans
    with (l':=(nodup_cancel mono_eq_dec (substM s (nodup var_eq_dec
    (m ++ n))))). apply nodup_cancel_Permutation. apply substM_Permutation.
    unfold make_mono. rewrite ← Permutation_VarSort_l. auto.
  induction m.
  - simpl. pose (mulPP_1r (substM s n)). rewrite mulPP_comm in e.
    pose (substM_is_poly s n). apply e in i. rewrite mulPP_mulPP''' in i.
    unfold mulPP''' in i. rewrite ← no_make_poly in i; auto.
    apply Permutation_sort_eq in i. rewrite i. rewrite no_nodup_NoDup.
    rewrite no_map_make_mono. auto. intros m Hin. apply (substM_is_poly s n);
    auto. apply NoDup_VarSorted. auto.
  - simpl substM at 2. apply Permutation_sort_eq. rewrite make_poly_refold.
    rewrite mulPP'''_refold. rewrite ← mulPP_mulPP'''. rewrite mulPP_assoc.
    repeat rewrite mulPP_mulPP'''. apply Permutation_sort_eq.
    rewrite substM_nodup_pointless; auto. simpl. rewrite mulPP_mulPP'''.
    unfold mulPP''' at 1. apply Permutation_sort_eq in IHm.
    rewrite make_poly_refold in IHm. rewrite mulPP'''_refold in IHm.
    rewrite no_nodup_cancel_NoDup in IHm. rewrite no_sort_MonoSorted in IHm.
    rewrite ← substM_nodup_pointless; auto. rewrite IHm. unfold make_poly.
    apply Permutation_trans with (l':=(nodup_cancel mono_eq_dec (nodup_cancel
      mono_eq_dec (map make_mono (concat (map (mulMP'' (substV s a))
      (mulPP''' (substM s m) (substM s n)))))))).
      apply nodup_cancel_Permutation. rewrite ← Permutation_MonoSort_l. auto.
      rewrite no_nodup_cancel_NoDup; auto.
    apply NoDup_nodup_cancel. apply substM_is_poly. apply NoDup_MonoSorted.
    apply substM_is_poly.
  - intros m0 Hin. apply (substM_is_poly s n). auto.
  - intros m0 Hin. apply (substM_is_poly s m). auto.
  - intros m0 Hin. apply (substM_is_poly s (make_mono (m ++ n))). auto.
Qed.

Lemma map_substM_distr_map_mulMP : ∀ m p s,
  is_poly_subst s →
  is_poly p →
  Permutation
    (nodup_cancel mono_eq_dec (map make_mono (concat (map (substM s) (map
      make_mono (mulMP'' p m))))))
    (nodup_cancel mono_eq_dec (map make_mono (concat (map (mulMP'' (map
      make_mono (concat (map (substM s) p)))) (map make_mono (substM s m)))))).
Proof.
  intros m p s Hps H. unfold mulMP'' at 1. apply Permutation_trans with (l':=
```

```
    (nodup_cancel mono_eq_dec (map make_mono (concat (map (substM s) (map
    make_mono (nodup_cancel mono_eq_dec (map make_mono (map (app m) p)))))))).
    apply nodup_cancel_Permutation, Permutation_map, Permutation_concat,
    Permutation_map, Permutation_map. unfold make_poly.
    rewrite ← Permutation_MonoSort_l. auto.
  apply Permutation_trans with (l':=(nodup_cancel mono_eq_dec (map make_mono
    (concat (map (substM s) (map make_mono (map make_mono (map (app m)
    (p))))))))). repeat rewrite List.concat_map. rewrite map_map.
    rewrite map_map. rewrite (map_map _ (map make_mono)).
    rewrite (map_map make_mono). rewrite nodup_cancel_concat_map. auto.
    intros x. rewrite no_map_make_mono. apply NoDup_MonoSorted.
    apply (substM_is_poly s (make_mono x)). intros m0 Hin.
    pose (substM_is_poly s (make_mono x)). apply i. auto.
  induction p; simpl.
  - induction (map make_mono (substM s m)); auto.
  - rewrite map_app. apply Permutation_sym. apply Permutation_trans with (l':=
      (nodup_cancel mono_eq_dec (map make_mono (concat (map (mulMP'' (map
      make_mono (substM s m))) (map make_mono (substM s a ++ concat (map
      (substM s) p)))))))). apply Permutation_sort_eq. repeat (rewrite
      make_poly_refold, mulPP'''_refold, ← mulPP_mulPP'''). apply mulPP_comm.
    repeat rewrite map_app. rewrite concat_app, map_app. apply Permutation_sym.
    apply nodup_cancel_app_Permutation. apply substM_distr_mulMP; auto. apply H.
    intuition. apply Permutation_sym. apply Permutation_trans with (l':=
      (nodup_cancel mono_eq_dec (map make_mono (concat (map (mulMP'' (map
      make_mono (concat (map (substM s) p)))) (map make_mono (substM s m))))))).
      apply Permutation_sort_eq. repeat (rewrite make_poly_refold,
      mulPP'''_refold, ← mulPP_mulPP'''). apply mulPP_comm.
    apply Permutation_sym. apply IHp. apply poly_cons in H. apply H.
Qed.
```

Here is the formulation of substitution distributing over polynomial multiplication. Similar to the proof for addition, it is very dense and makes common use of permutation facts. Where it differs from that proof is that it relies on the commutativity of multiplication. The proof of distribution over addition didn't need any properties of addition.

```
Lemma substP_distr_mulPP : ∀ p q s,
  is_poly_subst s →
  is_poly p →
  substP s (mulPP p q) = mulPP (substP s p) (substP s q).
Proof.
  intros p q s Hps H. repeat rewrite mulPP_mulPP'''. unfold substP, mulPP'''.
  apply Permutation_sort_eq. apply Permutation_trans with (l':=(nodup_cancel
    mono_eq_dec (map make_mono (concat (map (substM s) (nodup_cancel mono_eq_dec
    (map make_mono (concat (map (mulMP'' p) q))))))))).
```

```
    apply nodup_cancel_Permutation. apply Permutation_map.
    apply Permutation_concat. apply Permutation_map. unfold make_poly.
    rewrite ← Permutation_MonoSort_l. auto.
  apply Permutation_sym. apply Permutation_trans with (l':=(nodup_cancel
    mono_eq_dec (map make_mono (concat (map (mulMP'' (make_poly (concat (map
    (substM s) p)))) (nodup_cancel mono_eq_dec (map make_mono (concat (map
    (substM s) q))))))))). apply nodup_cancel_Permutation.
    apply Permutation_map. apply Permutation_concat. apply Permutation_map.
    unfold make_poly. rewrite ← Permutation_MonoSort_l. auto.
  apply Permutation_trans with (l':=(nodup_cancel mono_eq_dec (map make_mono
    (concat (map (mulMP'' (make_poly (concat (map (substM s) p)))) (map
    make_mono(concat (map (substM s) q)))))))). repeat rewrite (List.concat_map
    make_mono (map (mulMP'' _) _)). repeat rewrite (map_map _ (map make_mono)).
    apply nodup_cancel_concat_map. intros x. rewrite no_map_make_mono.
    unfold mulMP''. apply NoDup_MonoSorted. apply make_poly_is_poly.
    intros m Hin. apply mono_in_make_poly in Hin; auto.
  apply Permutation_sort_eq. rewrite make_poly_refold. rewrite mulPP'''_refold.
  rewrite ← mulPP_mulPP'''. rewrite mulPP_comm. rewrite mulPP_mulPP'''.
  apply Permutation_sort_eq. apply Permutation_trans with (l':=(nodup_cancel
    mono_eq_dec (map make_mono (concat (map (mulMP'' (map make_mono (concat (map
    (substM s) q)))) (nodup_cancel mono_eq_dec (map make_mono (concat (map
    (substM s) p))))))))). apply nodup_cancel_Permutation.
    apply Permutation_map. apply Permutation_concat. apply Permutation_map.
    unfold make_poly. rewrite ← Permutation_MonoSort_l. auto.
  apply Permutation_trans with (l':=(nodup_cancel mono_eq_dec (map make_mono
    (concat (map (mulMP'' (map make_mono (concat (map (substM s) q)))) (map
    make_mono (concat (map (substM s) p)))))))). repeat rewrite (List.concat_map
    make_mono (map (mulMP'' _) _)). repeat rewrite (map_map _ (map make_mono)).
    apply nodup_cancel_concat_map. intros x. rewrite no_map_make_mono.
    unfold mulMP''. apply NoDup_MonoSorted. apply make_poly_is_poly.
    intros m Hin. apply mono_in_make_poly in Hin; auto.
  apply Permutation_sort_eq. rewrite make_poly_refold. rewrite mulPP'''_refold.
  rewrite ← mulPP_mulPP'''. rewrite mulPP_comm. rewrite mulPP_mulPP'''.
  apply Permutation_sort_eq. apply Permutation_sym.
  apply Permutation_trans with (l':=(nodup_cancel mono_eq_dec (map make_mono
    (concat (map (substM s) (map make_mono (concat (map (mulMP'' p) q)))))))).
    repeat rewrite (List.concat_map make_mono (map _ _)).
    repeat rewrite map_map. rewrite nodup_cancel_concat_map. auto. intros x.
    rewrite no_map_make_mono. apply NoDup_MonoSorted; apply substM_is_poly.
    intros m Hin; apply (substM_is_poly s x); auto.
  induction q; auto. simpl. repeat rewrite map_app. repeat rewrite concat_app.
  repeat rewrite map_app. repeat rewrite ← (nodup_cancel_pointless (map _ _)).
```

```
  repeat rewrite ← (nodup_cancel_pointless_r _ (map _ _)).
  apply nodup_cancel_Permutation. apply Permutation_app.
  apply map_substM_distr_map_mulMP; auto. apply IHq.
Qed.
```

## 7.4 Unifiable Definitions

The following six definitions are all predicate functions that verify some property about substitutions or polynomials.

A *unifier* for a given polynomial $p$ is a substitution $s$ such that $s(p)\downarrow_P= 0$. This definition also includes that the range of the substitution only contain terms in polynomial form.

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

A polynomial $p$ is *unifiable* if there exists a unifier for $p$.

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

A substitution $u$ is a *composition* of two substitutions $s$ and $t$ if $u(x)\downarrow_P= t(s(x))\downarrow_P$ for every variable $x$. The lemma subst_comp_poly below extends this definition from variables to polynomials.

```
Definition subst_comp (s t u : subst) : Prop :=
  ∀ x,
  substP t (substP s [[x]]) = substP u [[x]].
```

A substitution $s$ is *more general* than a substitution $t$ if there exists a third substitution $u$ such that $t$ is a composition of $u$ and $s$.

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

Given a polynomial $p$, a substitution $s$ is the *most general unifier* of $p$ if $s$ is more general than every unifier of $p$.

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

Given a polynomial $p$, a substitution $s$ is a *reproductive unifier* of $p$ if $t$ is a composition of itself and $s$ for every unifier $t$ of $p$. This property is similar but stronger than most general because the substitution that composes with $s$ is restricted to $t$, whereas in most general it can be any substitution.

```
Definition reprod_unif (s : subst) (p : poly) : Prop :=
  unifier s p ∧
```

```
  ∀ t,
  unifier t p →
  subst_comp s t t.
```

Because the notion of most general is weaker than reproductive, it can be proven to logically follow as shown below. Any unifier that is reproductive is also most general.

```
Lemma reprod_is_mgu : ∀ p s,
  reprod_unif s p →
  mgu s p.
Proof.
  unfold mgu, reprod_unif, more_general, subst_comp.
  intros p s [].
  split; auto.
  intros.
  ∃ t.
  intros.
  apply H0; auto.
Qed.
```

As stated earlier, substitution composition can be extended to polynomials. This comes from the implicit fact that if two substitutions agree on all variables then they agree on all terms.

```
Lemma subst_comp_poly : ∀ s t u,
  is_poly_subst s →
  is_poly_subst t →
  is_poly_subst u →
  (∀ x, substP t (substP s [[x]]) = substP u [[x]]) →
  ∀ p,
  substP t (substP s p) = substP u p.
Proof.
  intros. induction p; auto. simpl. unfold substP at 2. simpl.
  rewrite ← make_poly_pointless_r. rewrite addPP_refold.
  rewrite substP_distr_addPP; auto. unfold substP at 3. simpl.
  rewrite ← make_poly_pointless_r. rewrite addPP_refold. f_equal.
  - induction a; auto. simpl. rewrite substP_distr_mulPP; auto. f_equal; auto.
    + rewrite ← subst_var_eq; auto. rewrite ← subst_var_eq; auto.
    + apply substV_is_poly; auto.
  - rewrite substP_refold. apply IHp.
Qed.
```

The last lemmas of this section state that the identity substitution is a reproductive unifier of the constant zero. Therefore it is also most general.

```
Lemma empty_unifier : unifier [] [].
Proof.
```

```
        unfold unifier, is_poly_subst. split; auto.
        intros. inversion H.
Qed.

Lemma empty_reprod_unif : reprod_unif [] [].
Proof.
  unfold reprod_unif, more_general, subst_comp.
  split; auto. apply empty_unifier.
Qed.

Lemma empty_mgu : mgu [] [].
Proof.
  apply reprod_is_mgu. apply empty_reprod_unif.
Qed.
```

# Chapter 8

# Library B_Unification.sve

```
Require Import List.
Import ListNotations.
Require Import Arith.
Require Import Permutation.
Require Export poly_unif.
```

## 8.1 Introduction

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 are generated until we are left with either of two problems; $1 \stackrel{?}{\approx}_B 0$ or $0 \stackrel{?}{\approx}_B 0$. 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.

## 8.2 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 $t \downarrow_P$. 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 $t \downarrow_P$ 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 in $t \downarrow_P$ that do not contain $x$. The original term can then be written as $x * q + r$.

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.

The function has_var determines whether a variable appears in a monomial.

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

The function elim_var removes a variable from each monomial in a polynomial. It is possible that this leaves the term not in polynomial form so it is then repaired with make_poly.

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

The function div_by_var produces a quotient $q$ and remainder $r$ from a polynomial $p$ and variable $x$ such that $p \approx_B x * q + r$ and $x$ does not occur in $r$.

```
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 $p \approx_B x * q + r$. This fact turns out to be rather hard to prove and needs the help of 10 or so other sudsidiary lemmas.

After eliminating a variable $x$ from a polynomial to produdce $r$, $x$ does not occur in $r$.

```
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.
  unfold make_poly in H.
  rewrite ← H in H0.
  apply In_sorted in H0.
  apply nodup_cancel_in in H0.
  rewrite map_map in H0.
  apply in_map_iff in H0 as [n []].
  rewrite ← H0.
  intro.
  rewrite make_mono_In in H2.
  apply remove_In in H2.
  auto.
Qed.
```

Eliminating a variable from a polynomial produces a term in polynomial form.

```
Lemma elim_var_is_poly : ∀ x p,
  is_poly (elim_var x p).
```

```
Proof.
  intros.
  unfold elim_var.
  apply make_poly_is_poly.
Qed.
```

Hint Resolve *elim_var_is_poly*.

The next four lemmas deal with the following scenario: Let $p$ be a term in polynomial form, $x$ be a variable that occurs in each monomial of $p$, and $r =$ elim_var $x$ $p$.

The term $r$ is a permutation of removing $x$ from $p$. Another way of looking at this statement is when elim_var repairs the term produced from removing a variable it only sorts that term.

```
Lemma elim_var_map_remove_Permutation : ∀ p x,
  is_poly p →
  (∀ m, In m p → In x m) →
  Permutation (elim_var x p) (map (remove var_eq_dec x) p).
Proof.
  intros p x H H0. destruct p as [|a p].
  - simpl. unfold elim_var, make_poly, MonoSort.sort. auto.
  - simpl. unfold elim_var. simpl. unfold make_poly.
    rewrite ← Permutation_MonoSort_l. rewrite unsorted_poly; auto.
    + rewrite ← map_cons. apply NoDup_map_remove; auto.
    + apply poly_cons in H. intros m Hin. destruct Hin.
      × rewrite ← H1. apply remove_is_mono. apply H.
      × apply in_map_iff in H1 as [y []]. rewrite ← H1. apply remove_is_mono.
        destruct H. unfold is_poly in H. destruct H. apply H4. auto.
Qed.
```

The term $(x * r) \downarrow_P$ is a permutation of the result of removing $x$ from $p$, appending $x$ to the end of each monomial, and repairing each monomial. The proof relies on the mulPP_map_app_permutation lemma from the poly library, which has a simpler goal but does much of the heavy lifting.

```
Lemma rebuild_map_permutation : ∀ p x,
  is_poly p →
  (∀ m, In m p → In x m) →
  Permutation (mulPP [[x]] (elim_var x p))
              (map (fun a ⇒ make_mono (a ++ [x]))
                   (map (remove var_eq_dec x) p)).
Proof.
  intros p x H H0. apply mulPP_map_app_permutation; auto.
  - apply (elim_var_not_in_rem x p); auto.
  - apply elim_var_map_remove_Permutation; auto.
Qed.
```

The term $p$ is a permutation of $(x * r) \downarrow_P$. Proof of this fact relies on the lengthy map_app_remove_Permutation lemma from poly.

```
Lemma elim_var_permutation : ∀ p x,
  is_poly p →
  (∀ m, In m p → In x m) →
  Permutation p (mulPP [[x]] (elim_var x p)).
Proof.
  intros p x H H0. pose (rebuild_map_permutation p x H H0).
  apply Permutation_sym in p0.
  pose (map_app_remove_Permutation p x H H0).
  apply (Permutation_trans p1 p0).
Qed.
```

Finally, $p = (x * r) \downarrow_P$.

```
Lemma elim_var_mul : ∀ x p,
  is_poly p →
  (∀ m, In m p → In x m) →
  p = mulPP [[x]] (elim_var x p).
Proof.
  intros. apply Permutation_Sorted_eq.
  - apply elim_var_permutation; auto.
  - unfold is_poly in H. apply Sorted_MonoSorted. apply H.
  - pose (mulPP_is_poly [[x]] (elim_var x p)). unfold is_poly in i.
    apply Sorted_MonoSorted. apply i.
Qed.
```

The function has_var is an equivalent boolean version of the In predicate.

```
Lemma has_var_eq_in : ∀ x m,
  has_var x m = true ↔ In x m.
Proof.
  intros.
  unfold has_var.
  rewrite existsb_exists.
  split; intros.
  - destruct H as [x0 []].
    apply Nat.eqb_eq in H0.
    rewrite H0. apply H.
  - ∃ x. rewrite Nat.eqb_eq. auto.
Qed.
```

Let a polynomial $p$ be partitioned by has_var $x$ into two sets $qx$ and $r$. Obviously, every monomial in $qx$ contains $x$ and no monomial in $r$ contains $x$.

```
Lemma part_var_eq_in : ∀ x p qx r,
  partition (has_var x) p = (qx, r) →
```

```
  ((∀ m, In m qx → In x m) ∧
   (∀ m, In m r → ¬ In x m)).
Proof.
  intros.
  split; intros.
  - apply part_fst_true with (a:=m) in H.
    + apply has_var_eq_in. apply H.
    + apply H0.
  - apply part_snd_false with (a:=m) in H.
    + rewrite ← has_var_eq_in. rewrite H. auto.
    + apply H0.
Qed.
```

The function div_by_var produces two terms both in polynomial form.

```
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.
  intros.
  unfold div_by_var in H0.
  destruct (partition (has_var x) p) eqn:Hpart.
  apply (part_is_poly _ _ _ _ H) in Hpart as Hp.
  destruct Hp as [Hpl Hpr].
  injection H0. intros Hr Hq.
  rewrite Hr in Hpr.
  apply part_var_eq_in in Hpart as [Hin Hout].
  split.
  - rewrite ← Hq; auto.
  - apply Hpr.
Qed.
```

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

```
Lemma div_eq : ∀ x p q r,
  is_poly p →
  div_by_var x p = (q, r) →
  p = addPP (mulPP [[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 _ _ _ _ HP HD).
  assert (is_poly qx ∧ is_poly r0) as [HPqx HPr0].
  apply (part_is_poly _ _ _ _ HP Hqr).
  rewrite ← Hq.
  rewrite ← (elim_var_mul x qx HPqx HIH).
  apply (part_add_eq (has_var x) _ _ _ HP).
  rewrite ← Hr.
  apply Hqr.
Qed.
```

Given a variable $x$, div_by_var produces two polynomials neither of which contain $x$.

```
Lemma div_var_not_in_qr : ∀ x p q r,
  div_by_var x p = (q, r) →
  ((∀ m, In m q → ¬ In x m) ∧
   (∀ m, In m r → ¬ In x m)).
Proof.
  intros.
  unfold div_by_var in H.
  assert (∃ qxr, qxr = partition (has_var x) p) as [[qx r0] Hqxr]. eauto.
  rewrite ← Hqxr in H.
  injection H. intros Hr Hq.
  split.
  - apply (elim_var_not_in_rem _ _ _ Hq).
  - rewrite Hr in Hqxr.
    symmetry in Hqxr.
    intros. intro.
    apply has_var_eq_in in H1.
    apply Bool.negb_false_iff in H1.
    revert H1.
    apply Bool.eq_true_false_abs.
    apply Bool.negb_true_iff.
    revert m H0.
    apply (part_snd_false _ _ _ _ _ Hqxr).
Qed.
```

This helper function build_poly is used to construct $p' = ((q + 1) * r) \downarrow_P$ given the two polynomials $q$ and $r$ as input.

```
Definition build_poly (q r : poly) : poly :=
```

```
  mulPP (addPP [[]] q) r.
```

The function build_poly produces a term in polynomial form.

```
Lemma build_poly_is_poly : ∀ q r,
  is_poly (build_poly q r).
Proof.
  unfold build_poly. auto.
Qed.

Hint Resolve build_poly_is_poly.
```

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

```
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 [Hps 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 ← (mulPP_0 (substP s q1)).
  rewrite ← Hsp0.
  rewrite Hp, Hq1.
  rewrite ← substP_distr_mulPP; auto.
  f_equal.
  apply (div_is_poly x p q r HPp) in HD.
  destruct HD as [HPq HPr].
  rewrite mulPP_addPP_1; auto.
Qed.
```

Given a polynomial $p$ and a variable $x$, div_by_var produces two polynomials $q$ and $r$ that have no more variables than $p$ has. Obviously, $q$ and $r$ don't contain $x$ either.

```
Lemma incl_div : ∀ x p q r xs,
  is_poly p →
  div_by_var x p = (q, r) →
  incl (vars p) (x :: xs) →
  incl (vars q) xs ∧ incl (vars r) xs.
```

```
Proof.
  intros. assert (Hdiv:=H0). unfold div_by_var in H0.
  destruct partition as [qx r0] eqn:Hpart. apply partition_Permutation in Hpart.
  apply Permutation_incl in Hpart as []. inversion H0. clear H2.
  assert (incl (vars q) (vars p)). unfold incl, vars in *. intros a Hin.
    apply nodup_In. apply nodup_In in Hin. apply In_concat_exists in Hin.
    destruct Hin as [m []]. rewrite <- H5 in H2. unfold elim_var in H2.
    apply In_sorted in H2. apply nodup_cancel_in in H2. rewrite map_map in H2.
    apply in_map_iff in H2. destruct H2 as [mx []]. rewrite <- H2 in H4.
    rewrite make_mono_In in H4. apply In_remove in H4. apply In_concat_exists.
    exists mx. split; auto. apply H3. intuition.
  assert (incl (vars r) (vars p)). rewrite H6 in H3. unfold incl, vars in *.
    intros a Hin. apply nodup_In. apply nodup_In in Hin.
    apply In_concat_exists in Hin. destruct Hin as [l []].
    apply In_concat_exists. exists l. split; auto. apply H3. intuition.
  split.
  - rewrite H5. apply incl_tran with (n:=(x::xs)) in H2; auto.
    apply incl_not_in in H2; auto. apply div_var_not_in_qr in Hdiv as [Hq _].
    apply in_mono_in_vars in Hq. auto.
  - apply incl_tran with (n:=(x::xs)) in H4; auto.
    apply incl_not_in in H4; auto. apply div_var_not_in_qr in Hdiv as [_ Hr].
    apply in_mono_in_vars in Hr. auto.
Qed.
```

Given a term $p$ decomposed into the form $(x * q + r) \downarrow_P$, then the polynomial $p' = ((q + 1) * r) \downarrow_P$ has no more variables than $p$ and does not contain $x$.

```
Lemma div_vars : forall x xs p q r,
  is_poly p ->
  incl (vars p) (x :: xs) ->
  div_by_var x p = (q, r) ->
  incl (vars (build_poly q r)) xs.
Proof.
  intros x xs p q r H Hincl Hdiv. unfold build_poly.
  apply div_var_not_in_qr in Hdiv as Hin. destruct Hin as [Hinq Hinr].
  apply in_mono_in_vars in Hinq. apply in_mono_in_vars in Hinr.
  apply incl_vars_mulPP. apply (incl_div _ _ _ _ _ H Hdiv) in Hincl. split.
  - apply incl_vars_addPP; auto. apply div_is_poly in Hdiv as []; auto. split.
    + unfold vars. simpl. unfold incl. intros a [].
    + apply Hincl.
  - apply Hincl.
Qed.

Hint Resolve div_vars.
```

## 8.3 Building Substitutions

This section handles how a solution is built from subproblem solutions. Given that term $p$ decomposed into $(x * q + r) \downarrow_P$ and $p' = ((q + 1) * r) \downarrow_P$, the lemma reprod_build_subst states that if some substitution $\sigma$ is a reproductive unifier of $p' \stackrel{?}{\approx}_B 0$, then we can build a substitution $\sigma'$ which is a reproductive unifier of $p \stackrel{?}{\approx}_B 0$. The way $\sigma'$ is built from $\sigma$ is defined in build_subst. Another replacement is added to $\sigma$ of the form $\{x \mapsto (x*(\sigma(q)+1)+\sigma(r)) \downarrow_P\}$ to construct $\sigma'$.

```
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 (mulPP [[x]] q1s) rs) in
  xs :: s.
```

The function build_subst produces a substitution whose range only contains polynomials.

```
Lemma build_subst_is_poly : ∀ s x q r,
  is_poly_subst s →
  is_poly_subst (build_subst s x q r).
Proof.
  unfold build_subst.
  unfold is_poly_subst.
  intros.
  destruct H0.
  - inversion H0. auto.
  - apply (H x0). auto.
Qed.
```

Given that term $p$ decomposed into $(x * q + r) \downarrow_P$, $p' = ((q + 1) * r) \downarrow_P$, and $\sigma$ is a reproductive unifier of $p' \stackrel{?}{\approx}_B 0$, then the substitution $\sigma'$ built from $\sigma$ unifies $p \stackrel{?}{\approx}_B 0$.

```
Lemma build_subst_is_unif : ∀ x p q r s,
  is_poly p →
  div_by_var x p = (q, r) →
  reprod_unif s (build_poly q r) →
  unifier (build_subst s x q r) p.
Proof.
  unfold reprod_unif, unifier.
  intros x p q r s Hpoly Hdiv [[Hps Hunif] Hreprod].
  assert (is_poly_subst (build_subst s x q r)).
    apply build_subst_is_poly; auto.
  split; auto.
  unfold build_poly in Hunif.
```

```
assert (Hnqr := Hdiv).
apply div_var_not_in_qr in Hnqr.
destruct Hnqr as [Hnq Hnr].
assert (HpolyQR := Hdiv).
apply div_is_poly in HpolyQR as [HpolyQ HpolyR]; auto.
apply div_eq in Hdiv; auto.
rewrite Hdiv.
rewrite substP_distr_addPP; auto.
rewrite substP_distr_mulPP; auto.
unfold build_subst.
rewrite (substP_cons _ _ Hnq).
rewrite (substP_cons _ _ Hnr).
assert (Hsx: (substP
      ((x,
       addPP
         (mulPP [[x]]
 (substP s (addPP [[]] q)))
         (substP s r)) :: s)
      [[x]]) = (addPP
       (mulPP [[x]]
 (substP s (addPP [[]] q)))
       (substP s r))).
  unfold substP. simpl.
  rewrite <- beq_nat_refl.
  rewrite mulPP_1r; auto. rewrite app_nil_r.
  rewrite no_make_poly; auto.
rewrite Hsx.
rewrite substP_distr_addPP; auto.
rewrite substP_1.
rewrite mulPP_distr_addPPr; auto.
rewrite mulPP_1r; auto.
rewrite mulPP_distr_addPP; auto.
rewrite mulPP_distr_addPP; auto.
rewrite mulPP_assoc.
rewrite mulPP_p_p; auto.
rewrite addPP_p_p; auto.
rewrite addPP_0; auto.
rewrite <- substP_distr_mulPP; auto.
rewrite <- substP_distr_addPP; auto.
rewrite <- (mulPP_1r r) at 2; auto.
rewrite mulPP_comm; auto.
rewrite (mulPP_comm r [[]]); auto.
```

```
  rewrite ← mulPP_distr_addPP; auto.
  rewrite addPP_comm; auto.
Qed.
```

Given that term $p$ decomposed into $(x * q + r) \downarrow_P$, $p' = ((q + 1) * r) \downarrow_P$, and $\sigma$ is a reproductive unifier of $p' \overset{?}{\approx}_B 0$, then the substitution $\sigma'$ built from $\sigma$ is reproductive with regards to unifiers of $p \overset{?}{\approx}_B 0$.

```
Lemma build_subst_is_reprod : ∀ x p q r s,
  is_poly p →
  div_by_var x p = (q, r) →
  reprod_unif s (build_poly q r) →
  ∀ t, unifier t p →
            subst_comp (build_subst s x q r) t t.
Proof.
  unfold reprod_unif.
  intros x p q r s HpolyP Hdiv [[HpsS HunifS] Hsub_comp] t HunifT.
  assert (HunifT' := HunifT).
  destruct HunifT as [HpsT HunifT].
  apply (div_build_unif _ _ _ _ _ HpolyP Hdiv) in HunifT'.
  unfold subst_comp in *.
  intros y.
  destruct (y =? x) eqn:Hyx.
  - unfold build_subst.
    assert (H: (substP ((x, addPP (mulPP [[x]] (substP s (addPP [[]] q)))
                                    (substP s r)) :: s) [[y]]) =
             (addPP (mulPP [[x]] (substP s (addPP [[]] q))) (substP s r))).
      unfold substP. simpl.
      rewrite Hyx.
      rewrite mulPP_1r; auto. rewrite app_nil_r.
      rewrite no_make_poly; auto.
    rewrite H.
    rewrite substP_distr_addPP; auto.
    rewrite substP_distr_mulPP; auto.
    pose (div_is_poly _ _ _ _ HpolyP Hdiv); destruct a.
    rewrite substP_distr_addPP; auto.
    rewrite substP_distr_addPP; auto.
    rewrite substP_1.
    assert (Hdiv2 := Hdiv).
    apply div_eq in Hdiv; auto.
    apply div_is_poly in Hdiv2 as [HpolyQ HpolyR]; auto.
    rewrite (subst_comp_poly s t t); auto.
    rewrite (subst_comp_poly s t t); auto.
```

```
    rewrite mulPP_comm; auto.
    rewrite mulPP_distr_addPP; auto.
    rewrite mulPP_comm; auto.
    rewrite mulPP_1r; auto.
    rewrite (addPP_comm (substP t [[x]]) _); auto.
    rewrite addPP_assoc; auto.
    rewrite (addPP_comm (substP t [[x]]) _ ); auto.
    rewrite ← addPP_assoc; auto.
    rewrite ← substP_distr_mulPP; auto.
    rewrite ← substP_distr_addPP; auto.
    rewrite mulPP_comm; auto.
    rewrite ← Hdiv.
    unfold unifier in HunifT.
    rewrite HunifT.
    rewrite addPP_0; auto.
    apply beq_nat_true in Hyx.
    rewrite Hyx.
    reflexivity.
  - unfold build_subst.
    rewrite substP_cons; auto.
    intros.
    inversion H; auto.
    rewrite ← H0.
    simpl. intro.
    destruct H1; auto.
    apply Nat.eqb_eq in H1.
    rewrite Hyx in H1.
    inversion H1.
Qed.
```

Given that term $p$ decomposed into $(x*q+r)\downarrow_P$, $p' = ((q+1)*r)\downarrow_P$, and a reproductive unifier $\sigma$ of $p' \overset{?}{\approx}_B 0$, then the substitution $\sigma'$ built from $\sigma$ is a reproductive unifier $p \overset{?}{\approx}_B 0$ based on the previous two lemmas.

```
Lemma reprod_build_subst : ∀ x p q r s,
  is_poly p →
  div_by_var x p = (q, r) →
  reprod_unif s (build_poly q r) →
  reprod_unif (build_subst s x q r) p.
Proof.
  intros. unfold reprod_unif. split.
  - apply build_subst_is_unif; auto.
  - apply build_subst_is_reprod; auto.
Qed.
```

## 8.4 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 $p = (x{*}q{+}r)\downarrow_P$ and $p' = ((q{+}1){*}r)\downarrow_P$, the solution to $p \overset{?}{\approx}_B 0$ is built from the solution to $p' \overset{?}{\approx}_B 0$. If $\sigma$ solves $p' \overset{?}{\approx}_B 0$, then $\sigma \cup \{x \mapsto (x * (\sigma(q) + 1) + \sigma(r))\downarrow_P\}$ solves $p \overset{?}{\approx}_B 0$.

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 at least one less variable.

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

The function sve simply calls sveVars with an initial fuel of vars $p$.

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

## 8.5 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 be not 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. The two cases of the proof are handled seperately by the lemmas sveVars_some and sveVars_none.

If sveVars produces a substitution $\sigma$, then the range of $\sigma$ only contains polynomials.

```
Lemma sveVars_poly_subst : ∀ xs p,
  incl (vars p) xs →
  is_poly p →
  ∀ s, sveVars xs p = Some s →
            is_poly_subst s.
Proof.
  induction xs as [|x xs]; intros.
  - simpl in H1. destruct p; inversion H1. unfold is_poly_subst.
    intros x p [].
  - intros.
    assert (∃ qr, div_by_var x p = qr) as [[q r] Hqr]. eauto.
    simpl in H1.
    rewrite Hqr in H1.
    destruct (sveVars xs (build_poly q r)) eqn:Hs0; inversion H1.
    apply IHxs in Hs0; eauto.
    apply build_subst_is_poly; auto.
Qed.
```

If sveVars produces a substitution $\sigma$ for the polynomial $p$, then $\sigma$ is a most general unifier of $p \overset{?}{\approx}_B 0$.

```
Lemma sveVars_some : ∀ (xs : list var) (p : poly),
  NoDup xs →
  incl (vars p) xs →
  is_poly p →
  ∀ s, sveVars xs p = Some s →
            mgu s p.
Proof.
  intros xs p Hdup H H0 s H1.
  apply reprod_is_mgu.
  revert xs p Hdup H H0 s H1.
  induction xs as [|x xs].
  - intros. simpl in H1. destruct p; inversion H1.
    apply empty_reprod_unif.
  - intros.
    assert (∃ qr, div_by_var x p = qr) as [[q r] Hqr]. eauto.
    simpl in H1.
    rewrite Hqr in H1.
```

```
    destruct (sveVars xs (build_poly q r)) eqn:Hs0; inversion H1.
    apply NoDup_cons_iff in Hdup as Hnin. destruct Hnin as [Hnin Hdup0].
    apply sveVars_poly_subst in Hs0 as HpsS0; eauto.
    apply IHxs in Hs0; eauto.
    apply reprod_build_subst; auto.
Qed.
```

If sveVars does not produce a substitution for the polynomial $p$, then the problem $p \overset{?}{\approx}_B 0$ is not unifiable.

```
Lemma sveVars_none : ∀ (xs : list var) (p : poly),
  NoDup xs →
  incl (vars p) xs →
  is_poly p →
  sveVars xs p = None →
  ¬ unifiable p.
Proof.
  induction xs as [|x xs].
  - intros p Hdup H H0 H1. simpl in H1. destruct p; inversion H1. intro.
    unfold unifiable in H2. destruct H2. unfold unifier in H2.
    apply incl_nil in H. apply no_vars_is_ground in H; auto.
    destruct H; inversion H.
    rewrite H4 in H2.
    rewrite H5 in H2.
    rewrite substP_1 in H2.
    inversion H2. inversion H6.
  - intros p Hdup H H0 H1.
    assert (∃ qr, div_by_var x p = qr) as [[q r] Hqr]. eauto.
    simpl in H1.
    rewrite Hqr in H1.
    destruct (sveVars xs (build_poly q r)) eqn:Hs0; inversion H1.
    apply NoDup_cons_iff in Hdup as Hnin. destruct Hnin as [Hnin Hdup0].
    apply IHxs in Hs0; eauto.
    unfold not, unifiable in *.
    intros.
    apply Hs0.
    destruct H2 as [s Hu].
    ∃ s.
    apply (div_build_unif x p); auto.
Qed.

Hint Resolve NoDup_vars incl_refl.
```

If sveVars produces a substitution $\sigma$ for the polynomial $p$, then $\sigma$ is a most general unifier of $p \overset{?}{\approx}_B 0$. Otherwise, $p \overset{?}{\approx}_B 0$ is not unifiable.

```
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.
  destruct (sveVars (vars p) p) eqn: Hsve.
  - apply (sveVars_some (vars p)); auto.
  - apply (sveVars_none (vars p)); auto.
Qed.
```

If sve produces a substitution $\sigma$ for the polynomial $p$, then $\sigma$ is a most general unifier of $p \overset{?}{\approx}_B 0$. Otherwise, $p \overset{?}{\approx}_B 0$ is not unifiable.

```
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.
```