Contents

Chapter 1

Library B_Unification.intro

1.1 Introduction

In the field of computational theory, one problem of significance is that of equational unification; namely, the finding of solutions to a given set of equations with respect to a set of relational axioms. While there are several variants of equational unification, for the purposes of this paper we are going to limit our scope to that of Boolean unification, which deals with the finding of unifiers for equations operating across Boolean rings. As any problem space would imply, there exists a great deal of research in the formal verification of unification algorithms; our research focused on two of these algorithms: Lowenheim's formula, and Succesive Variable Eliminaton. To conduct our research, we utilized the Coq proof assistant to create formal specifications of both of these algorithms' behaviors in addition to proving their correctness. While proofs for both of these algorithms already exist, prior to the writing of this paper, no formal treatment using a proof asssistant such as Coq had been undertaken, so it is hoped that our efforts provide yet another guarantee for the veritable correctness of these respective algorithms.

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 σ 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 σ *unifies* an equation $s \stackrel{?}{=} t$ if applying σ to both sides makes them equal $\sigma(s) = \sigma(t)$. If σ unifies every equation in the problem S, we call σ 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 σ is *more general* than σ' if there exists a third substitution δ 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 $\{f(x, a) \stackrel{?}{=} f(a, b)\}$ 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 $t \stackrel{?}{\approx}_B 0$. Given a problem in its normal form $\{s_1 \stackrel{?}{\approx}_B t_1, ..., s_n \stackrel{?}{\approx}_B t_n\}$, we can transform it into $\{s_1 + t_1 \stackrel{?}{\approx}_B 0, ..., s_n + t_n \stackrel{?}{\approx}_B 0\}$ 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 $\{t_1 \stackrel{?}{\approx}_B 0, ..., t_n \stackrel{?}{\approx}_B 0\}$, we can transform it into $\{(t_1 + 1) * ... * (t_n + 1) \stackrel{?}{\approx}_B 1\}$. 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.

More simply stated, formal verification is the process of examining whether a system or a theory "does what it is supposed to do." If it is a system, then scientistis formally verify that it satisfies its design requirements. Formal verification is also different to testing. Software testing is trying to detect "bugs" specific errors and requirements in the system, whereas verfification acts as a general safeguard that the system is always error-free. As Edsger Dijkstra stated, testing can be used to show the presence of bugs, but never to show their absence. If it a theory, scientists formally verify the correctness of the theory by formulating its proof using a formal language, axioms and inference rules.

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 requirements. 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

Formal verification is used in the industry to verify the correctness of software or hardware systems. When used to verify systems, formal verification can be thought as a mathematical proof of the correctness of a design with respect to a formal specification. The actual system is represented by a formal model and then the formal verification happens on the model, based on the required specifications of the system. Unlike testing, formal verification is exhaustive and imporves the understanding of a system. Howeverm, it is difficult to make for real-worls systems, time consuming and only as reliable as the actual model.

1.3.3 Verifying Theories

Formal verification is also used in to prove theorems. These theorems could be related to a computing system or just plain mathematical abstract theorems. As in proving systems, when proving theorems one also needs a formal logic to formulate the theorem and prove it. A formal logic consists of a formal languge to express the theorems, a collection of formulas called axioms and inference rules to derive new axioms based on existing ones. A theorem to be proven could be in a logical form, like DeMorgan's Law or it could in another mathematical area; in trigonometry for example, it could be useful to prove that sin(x + y) = sin(x) * cos(y) + cos(x) * sin(y) , formally, because that proof could be used as building block in a more complex system. Sometimes proving the corectness of a real world systems boils down to verifying mathemetical proofs like the previous one, so the two approaches are often linked together.

1.4 Importance

Given that the emergence of proof assistance software is still in its infancy relative to the age-old traditional methods of theorem proving, it would be a disservice for us to not establish the importance of this technology and its implications for the future of mathematics. Unlike in years past, where typos or subliminal edge cases could derail the developments of sound theorems, proof assistants now guarantee through their properties of verification that any development run by them is free from such lapses in logic on account of the natural failings of the human mind. Additionally, due to the adoption of a well-defined shared language, many of the ambiguities naturally present in the exchange of mathematical ideas between colleagues are mitigated, leading to a smoother learning curve for newcomers trying to understand the nuts and bolts of a complex theorem. The end result of these phenomenon is a faster iterative development cycle for mathemeticians as they now can spend more time on proving things

and building off of the work of others since they no longer need to devote as much of their efforts towards verifying the correctness of the theorems they are operating across.

Bearing this in mind, it should come as no surprise that there is a utility in going back to older proofs that have never been verified by a proof assistant and redeveloping them for the purposes of ensuring their correctness. If the theorem is truly sound, it stands to reason that any additional rigorous scrutiny would only serve to bolster the credibility of its claims, and conversely, if the theorem is not sound, it is a benefit to the academic community at large to be made aware of its shortcomings. Therefore, for these reasons we set out to formally verify two algorithms across Boolean Unification.

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 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.

Add Parametric Morphism : PRODUCT with
  signature eqv ==> eqv ==> eqv as PRODUCT_mor.

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.
```

```
Lemma x_equal_y_x_plus_y :
  ∀ x y, x == y ↔ x + y == T0.

Hint Resolve mul_x_x_plus_T1 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.

Lemma mul_id_sym :
  ∀ x, x × T1 == x.

Lemma mul_T0_x_sym :
  ∀ x, x × T0 == T0.

Lemma sum_assoc_opp :
 ∀ x y z, x + (y + z) == (x + y) + z.

Lemma mul_assoc_opp :
 ∀ x y z, x × (y × z) == (x × y) × z.

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

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.

Lemma vs_includes_false : ∀ (x : var) (lvar : list var),
  var_set_includes_var x lvar = false → ¬ In x lvar.

Lemma in_dup_and_non_dup : ∀ (x: var) (lvar : list var),
  In x lvar ↔ In x (var_set_create_unique lvar).
```

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].

Example var_set_is_unique_ex1 :
  var_set_is_unique [0;2;2;2] = false.

Example term_vars_ex1 :
  term_vars (VAR 0 + VAR 0 + VAR 1) = [0;0;1].

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

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.
```

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) = [].
```

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).
Example ex_gt2 :
  ground_term (VAR 0 × T1) → False.
```

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

In this subsection we make the fundamental definitions of substitutions, basic functions for them, accompanying lemmas and some propsitions.

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 to themselves.

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

Lemma sum_comm_compat t1 t2: ∀ (sigma: subst),
  apply_subst (t1 + t2) sigma == apply_subst (t2 + t1) sigma.
Hint Resolve sum_comm_compat.

Lemma sum_assoc_compat t1 t2 t3: ∀ (sigma: subst),
  apply_subst ((t1 + t2) + t3) sigma == apply_subst (t1 + (t2 + t3)) sigma.
Hint Resolve sum_assoc_compat.
```

```
Lemma sum_id_compat t: ∀ (sigma: subst),
  apply_subst (T0 + t) sigma == apply_subst t sigma.
Hint Resolve sum_id_compat.

Lemma sum_x_x_compat t: ∀ (sigma: subst),
  apply_subst (t + t) sigma == apply_subst T0 sigma.
Hint Resolve sum_x_x_compat.

Lemma mul_comm_compat t1 t2: ∀ (sigma: subst),
  apply_subst (t1 × t2) sigma == apply_subst (t2 × t1) sigma.
Hint Resolve mul_comm_compat.

Lemma mul_assoc_compat t1 t2 t3: ∀ (sigma: subst),
  apply_subst ((t1 × t2) × t3) sigma == apply_subst (t1 × (t2 × t3)) sigma.
Hint Resolve mul_assoc_compat.

Lemma mul_x_x_compat t: ∀ (sigma: subst),
  apply_subst (t × t) sigma == apply_subst t sigma.
Hint Resolve mul_x_x_compat.

Lemma mul_T0_x_compat t: ∀ (sigma: subst),
  apply_subst (T0 × t) sigma == apply_subst T0 sigma.
Hint Resolve mul_T0_x_compat.

Lemma mul_id_compat t: ∀ (sigma: subst),
  apply_subst (T1 × t) sigma == apply_subst t sigma.
Hint Resolve mul_id_compat.

Lemma distr_compat t1 t2 t3: ∀ (sigma: subst),
  apply_subst (t1 × (t2 + t3)) sigma ==
  apply_subst ((t1 × t2) + (t1 × t3)) sigma.
Hint Resolve distr_compat.

Lemma refl_comm_compat t1 t2: ∀ (sigma: subst),
  apply_subst t1 sigma == apply_subst t2 sigma →
  apply_subst t2 sigma == apply_subst t1 sigma.
Hint Resolve refl_comm_compat.

Lemma trans_compat t1 t2 t3 : ∀ (sigma: subst),
  apply_subst t1 sigma == apply_subst t2 sigma →
  apply_subst t2 sigma == apply_subst t3 sigma →
  apply_subst t1 sigma == apply_subst t3 sigma.
Hint Resolve trans_compat.

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

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

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

```
Lemma ground_term_cannot_subst : ∀ x,
  ground_term x →
  ∀ s, apply_subst x s == x.
```

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 : ∀ s x y,
  apply_subst x s + apply_subst y s == apply_subst (x + y) s.

Lemma subst_associative : ∀ s x y,
  apply_subst x s × apply_subst y s == apply_subst (x × y) s.

Lemma subst_sum_distr_opp : ∀ s x y,
  apply_subst (x + y) s == apply_subst x s + apply_subst y s.

Lemma subst_mul_distr_opp : ∀ s x y,
  apply_subst (x × y) s == apply_subst x s × apply_subst y s.

Lemma var_subst: ∀ (v : var) (ts : term),
  apply_subst (VAR v) [(v , ts)] == ts.
```

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.

Example subst_ex2 :
  apply_subst (VAR 0 × VAR 1) [(0, T0)] == T0.
```

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)].
```

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.
```

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)].
```

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.
```

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).
```

2.7 Most General Unifier

In this subsection we define propositions, lemmas and examples related to the most general unifier.

While the property of a term being unifiable is certainly important, it should come as no surprise that not all unifiers are created equal; in fact, certain unifiers possess the desirable property of being *more general* than others. For this reason, let us now formally define the

concept of a *most general unifier* (mgu): a unifier such that with respect to a given term, all other unifiers are instances of it, or in other words, less general than it.

The first step towards establishing the concept of a mgu requires us to formalize the notion of a unifier being more general than another. To accomplish this goal, let us formulate the definition of a substitution composing another one; or in other words, to say that a substitution is more general than another one.

```
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.
```

Now that we have articulated the concept of composing substitutions, let us now formulate the definition for a most general unifier.

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

While this definition of a most general unifier is certainly valid, it is a somewhat unwieldy formulation. For this reason, let us now define an alternative definition called a *reproductive unifier*, and then prove it to be equivalent to our definition of a most general unifier. This will make our proofs easier to formulate down the road as the task of proving a unifier to be reproductive is substantially easier than proving it to be most general directly.

```
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.

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

2.8 Auxilliary Computational Operations and Simplifications

These functions below will come in handy later during the Lowenheim formula proof. They mainly lay the groundwork for providing the computational nuts and bolts for Lowenheim's algorithm for finding most general unifiers.

```
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 ⇒ identical x x1 && identical y y1
  | PRODUCT x y, _ ⇒ false
  | SUM x y, SUM x1 y1 ⇒ identical x x1 && identical y y1
  | 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.

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

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

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

Being able to simplify a term can be a usefool tool. Being able to use the simplified version of the term as the equivalent version of the original term can also be useful since many of our functions simplify the term first.

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

Chapter 3

Library B_Unification.lowenheim_formula

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

3.1 Introduction

In this section we formulate Lowenheim's algorithm using the data structures and functions defined in the terms library. The final occuring main function Lowenheim_Main, takes as input a term and produces a substitution that unifies the given term and it is defined towards the end of the file. The substitution is said to be a most general unifier and not a mere substitution, but that statement is proven in the lowenheim_proof file. In this section we focus on the formulation of the algorithm itself, without any proofs about the properties of the formula or the algorithm.

3.2 Lowenheim's Builder

In this subsection we are implementing the main component of Lowenheim's algorithm, which is the "builder" of Lowenheim's substitution for a given term. This implementation strictly follows as close as possible the formal, mathematical format of Lowenheim's algorithm.

Here is a skeleton function for building a substition on the format $\sigma(x) := (s+1)*\sigma_1(x)+ s * \sigma_2(x)$, each variable of a given list of variables, a given term s and subtitutions σ_1 and σ_2. This skeleton function is a more general format of Lowenheim's builder.

```
Fixpoint build_on_list_of_vars (list_var : var_set) (s : term) (sig1 : subst)
                                (sig2 : subst) : subst :=
  match list_var with
```

```
| [] ⇒ []
| v' :: v ⇒ (v', (s + T1) × apply_subst (VAR v') sig1 +
                    s × apply_subst (VAR v') sig2)
                :: build_on_list_of_vars v s sig1 sig2
end.
```

This is the function to build a Lowenheim subsitution for a term t, given the term t and a unifier of t, using the previously defined skeleton function. The list of variables is the variables within t and the substitions are the identical subtitution and the unifer of the term. This fuction will often be referred in the rest of the document as our "Lownheim builder" or the "Lownheim substitution builder", etc.

```
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.
```

3.3 Lowenheim's Algorithm

In this subsection we enhance Lowenheim's builder to the level of a complete algorithm that is able to find ground substitutions before feeding them to the main formula to generate a most general unifier

3.3.1 Auxillary Functions and Definitions

This is a function to update a term, after it applies to it a given substitution and simplifies it.

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

Here is a function to determine if a term is the ground term T0.

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

In this development we have the need to be able to represent both the presence and the absence of a substitution. In case for example our find_unifier function cannot find a unifier for an input term, we need to be able to return a `subst` nil type, like a substitution option that states no substition was found. We are using the built-in Some and None inductive options (that are used as Some σ and None) to represent some substitution and no substition repsectively. The type of the two above is the inductive **option** {A:`type`} that can be attached to any type; in our case it is **option** `subst`.

Our Lownheim builder works when we provide an already existing unifier of the input term t. For our implementation to be complete we need to be able to generate that initial unifier ourselves. That is why we define a function to find a single ground unifier, recursively.

It finds a substitution with ground terms that makes the given input term equivalent to T0. To use it, start with an empty list of replacements as the input s : subst.

```
Fixpoint rec_subst (t : term) (vars : var_set) (s : subst) : subst :=
  match vars with
  | [] ⇒ s
  | v' :: v ⇒
    if (term_is_T0 (update_term (update_term t ((v' , T0) :: s))
                                (rec_subst (update_term t ((v' , T0) :: s))
                                           v ((v' , T0) :: s))))
    then rec_subst (update_term t ((v' , T0) :: s)) v ((v' , T0) :: s)
    else
      if (term_is_T0 (update_term (update_term t ((v' , T1) :: s))
                                  (rec_subst (update_term t ((v' , T1) :: s))
                                             v ((v' , T1) :: s))))
      then rec_subst (update_term t ((v' , T1) :: s)) v ((v' , T1) :: s)
      else rec_subst (update_term t ((v' , T0) :: s)) v ((v' , T0) :: s)
  end.
```

Next is a function to find a ground unifier of the input term, if it exists.

```
Fixpoint find_unifier (t : term) : option subst :=
  match update_term t (rec_subst t (term_unique_vars t) []) with
  | T0 ⇒ Some (rec_subst t (term_unique_vars t) [])
  | _ ⇒ None
  end.
```

Here is the main Lowenheim's formula; given a term, produce an MGU (a most general substitution that makes it equivalent to T0), if there is one. Otherwise, return None. This function is oftern referred in the rest of the document as "Lowenheim Main" function or "Main Lowenheim" function, etc.

```
Definition Lowenheim_Main (t : term) : option subst :=
  match find_unifier t with
  | Some s ⇒ Some (build_lowenheim_subst t s)
  | None ⇒ None
  end.
```

3.4 Lowenheim's Functions Testing

In this subsection we explore ways to test the correctness of our Lownheim's functions on specific inputs.

Here is a function to test the correctness of the output of the find_unifier helper function defined above. True means expected output was produced, false otherwise.

```
Definition Test_find_unifier (t : term) : bool :=
```

```
  match find_unifier t with
  | Some s ⇒ term_is_T0 (update_term t s)
  | None ⇒ true
  end.
```

Here is a function to apply Lowenheim's substitution on the term - the substitution produced by the Lowenheim main function.

```
Definition apply_lowenheim_main (t : term) : term :=
  match Lowenheim_Main t with
  | Some s ⇒ apply_subst t s
  | None ⇒ T1
  end.
```

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.
```

4.1 Introduction

In this chapter we provide a proof that our Lowenheim_Main function defined in lowenheim_formula provides a unifier that is most general. Our final top level proof (found at the end of this file) proves two statements: 1) If a term is unifiable, then our own defined Lowenheim_Main function produces a most general unifier (mgu). 2) If a term is not unifiable, then our own defined *Lownheim_Main* function produces a None substitution. We prove the above statements with a series of proofs and sub-groups of proofs that help us get to the final top-level statements mentioned above.

4.2 Auxillary Declarations and Their Lemmas Useful For the Final Proofs

In this section we provide definitions and proofs of helper functions, propositions, and lemmas that will be later used in other proofs.

This is the definition of an under_term. An under_term is a proposition, or a relationship between two terms. When a term t is an under_term of a term t' then each of the unique variables found within t are also found within the unique variables of t'.

```
Definition under_term (t : term) (t' : term) : Prop :=
  ∀ (x : var ),
  In x (term_unique_vars t) → In x (term_unique_vars t').
```

This is a simple lemma for under_terms that states that a term is an under_term of itself.

```
Lemma under_term_id : ∀ (t : term),
  under_term t t.
```

This is a lemma to prove the summation distribution property of the function term_vars: the term_vars of a sum of two terms is equal to the concantentation of the term_vars of each individual term of the original sum.

```
Lemma term_vars_distr : ∀ (t1 t2 : term),
  term_vars (t1 + t2) = term_vars t1 ++ term_vars t2.
```

This is a lemma to prove an intuitive statement: if a variable is within the term_vars (list of variables) of a term, then it is also within the term_vars of the sum of that term and any other term.

```
Lemma tv_h1: ∀ (t1 t2 : term) (x : var),
  In x (term_vars t1) → In x (term_vars (t1 + t2)).
```

This is a lemma similar to the previous one, to prove an intuitive statement: if a variable is within the term_vars (list of variables) of a term, then it is also within the term_vars of the sum of that term and any other term, but being added from the left side.

```
Lemma tv_h2: ∀ (t1 t2 : term) (x : var),
  In x (term_vars t2) → In x (term_vars (t1 + t2)).
```

This is a helper lemma for the under_term relationship : if the sum of two terms is a subterm of another term t', then the left component of the sum is also a subterm of the other term t'.

```
Lemma helper_2a: ∀ (t1 t2 t' : term),
  under_term (t1 + t2) t' → under_term t1 t'.
```

This is a helper lemma for the under_term relationship : if the sum of two terms is a subterm of another term t', then the right component of the sum is also a subterm of the other term t'.

```
Lemma helper_2b: ∀ (t1 t2 t' : term),
  under_term (t1 + t2) t' → under_term t2 t'.
```

This is a helper lemma for lists and their elements: if a variable is a member of a list, then it is equal to the first element of that list or it is a member of the rest of the elements of that list.

```
Lemma elt_in_list: ∀ (x: var) (a : var) (l : list var),
  In x (a :: l) →
  x = a ∨ In x l.
```

This is a similar lemma to the previous one, for lists and their elements: if a variable is not a member of a list, then it is not equal to the first element of that list and it is not a member of the rest of the elements of that list.

Lemma elt_not_in_list: $\forall$ (x: var) (a : var) (l : list var),
$\neg$ In x (a :: l) $\rightarrow$
$x \neq a \wedge \neg$ In x l.

This is a lemma for an intuitive statement for the variables of a term: a variable x belongs to the list of unique variables (term_unique_vars) found within the variable-term that is constructed by variable itself VAR x.

Lemma in_list_of_var_term_of_var: $\forall$ (x : var),
In x (term_unique_vars (VAR x)).

Lemma var_in_out_list: $\forall$ (x : var) ($lvar$: list var),
In x $lvar$ $\vee \neg$ In x $lvar$.

4.3 Proof That Lownheim's Algorithm Unifies a Given Term

In this section, we prove that our own defined Lowenheim builder from lowenheim_formula (build_lowenheim_subst), produces a unifier; that is, given unifiable term and one unifier of the term, it also produces another unifier of this term (and as explained in the terms library, a unifier is a substitution that when applied to term it produces a term equivalent to the ground term T0.

This is a helper lemma for the skeleton function defined in lowenheim_formula: If we apply a substitution on a term-variable VAR x, and that substitution is created by the skeleton function build_on_list_of_vars applied on a single input variable x, then the resulting term is equivalent to: the resuting term from applying a substitution on a term-variable VAR x, and that substitution being created by the skeleton function build_on_list_of_vars applied on an input list of variables that contains variable x.

Lemma helper1_easy: $\forall$ (x: var) ($lvar$: list var)
($sig1$ $sig2$: subst) (s : term),
In x $lvar$ $\rightarrow$
apply_subst (VAR x) (build_on_list_of_vars $lvar$ s $sig1$ $sig2$) ==
apply_subst (VAR x) (build_on_list_of_vars [x] s $sig1$ $sig2$).

This is another helper lemma for the skeleton function build_on_list_of_vars and it can be rephrased this way: applying two different substitutions on the same term-variable give the same result. One subsitution containing only one replacement, and for its own variable. The other subsitution contains the previous replacement but also more replacements for other variables (that are obviously not in the variables of our term-variable). So, the replacements for the extra variables do not affect the application of the subsitution - hence the resulting term.

```
Lemma helper_1: ∀ (t' s : term) (v : var) (sig1 sig2 : subst),
  under_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).
```

Lemma 10.4.5 from book X on page 254-255. This a very significant lemma used later for the proof that our Lownheim builder function (not the Main function, but the builder function), gives a unifier (not necessarily an mgu, which would be a next step of the proof). It states that if a term t is an under_term of another term t', then applying a substitution– a substitution created by giving the list of variables of term t' on the skeleton function *build_list_of_vars*–, on the term t, a term that has the same format: $(s+1)*\sigma_1(t)+s*\sigma_2(t)$ as the each replacements of each variable on any substitution created by skeleton function: $(s+1)*\sigma_1(x)+s*\sigma_2(x)$.

```
Lemma subs_distr_vars_ver2 : ∀ (t t' s : term) (sig1 sig2 : subst),
  under_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.
```

This is an intermediate lemma occuring by the previous lemma 10.4.5. Utilizing lemma 10.4.5 and also using two substitutions for the skeleton function *build_on_list_vars* gives a substituion the unifies the term; the two substitutions being a known unifier of the term and the identity substitution.

```
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.
```

This is the resulting lemma from this subsection: Our Lowenheim's subsitution builder produces a unifier for an input term; namely, a substitution that unifies the term, given that term is unifiable and we know an already existing unifier τ.

```
Lemma lownheim_unifies: ∀ (t : term) (tau : subst),
  unifier t tau →
  apply_subst t (build_lowenheim_subst t tau) == T0.
```

4.4 Proof That Lownheim's Algorithm Produces a Most General Unifier

In the previous section we proved that our Lowenheim builder produces a unifier, if we already know an existing unifier of the term. In this section we prove that that unifier is a

most general unifier.

4.4.1 Proof That Lownheim's Algorithm Produces a Reproductive Unifier

In this subsection we will prove that our Lowenheim builder gives a unifier that is reproductive; this will help us in the proof that the resulting unifier is an mgu, since a reproductive unifier is a "stronger" property than an mgu.

This is a lemma for an intuitive statement for the skeleton function *build_on_list_vars*: if a variable x is in a list l, and we apply a substitution created by the *build_on_list_vars* function given input list l, on the term-variable VAR x, then we get the replacement for that particular variable that was contained in the original substitution. So basically if build_on_list_of_vars is applied on a list of variables l $(x_1, x2, x_3, ..., x_n)$, then the resulting substitution is in the format $x_i \mapsto (s+1)*\sigma_1(x_i)+s*\sigma_2(x_i)$ for each x_i. If we apply that substitution on the term-variable x_1, *wewillgettheinitialformatofthereplacement* :(s + 1) \ast \sigma{1}(x{1}) + s \ast \sigma{2}(x{1}).*Itcanbethoughtas*"*reverseapplication*"*oftheskeletonfunction*.

```
Lemma lowenheim_rephrase1_easy : ∀ (l : list var) (x : var)
                                   (sig1 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.
```

This is a helper lemma for an intuitive statement: if a variable x is found in a list of variables l, then applying the subsitution created by the build_id_subst function given input list l, on the term-variable VAR x, we will get the same VAR x back.

```
Lemma helper_3a: ∀ (x: var) (l: list var),
  In x l →
  apply_subst (VAR x) (build_id_subst l) == VAR x.
```

This is a lemma for an intuitive statement for the Lowenheim builder, very similar to lemma lowenheim_rephrase1_easy: applying Lowenheim's subtitution given an input term t, on any term-variable of the term t, gives us the initial format of the replacement for that variable (Lowenheim's reverse application).

```
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.
```

This is a lemma for an intuitive statement for the skeleton function *build_on_list_vars* that resemebles a lot of lowenheim_rephrase1_easy: if a variable x is not in a list l, and we apply a substitution created by the *build_on_list_vars* function given input list l, on the term-variable VAR x, then we get the term-variable VAR x back; that is expected since the replacements in the substitution should not contain any entry with variable x.

```
Lemma lowenheim_rephrase2_easy : ∀ (l : list var) (x : var)
                                   (sig1 sig2 : subst) (s : term),
  ¬ (In x l) →
  apply_subst (VAR x) (build_on_list_of_vars l s sig1 sig2) ==
  VAR x.
```

This is a lemma for an intuitive statement for the Lowenheim builder, very similar to lemma lowenheim_rephrase2_easy and lowenheim_rephrase1: applying Lowenheim's subtitution given an input term t, on any term-variable not of the ones of term t, gives us back the same term-variable.

```
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.
```

This is the resulting lemma of the secton: our Lowenheim builder *build_lownheim_subst* gives a reproductive unifier.

```
Lemma lowenheim_reproductive: ∀ (t : term) (tau : subst),
  unifier t tau →
  reproductive_unifier t (build_lowenheim_subst t tau).
```

4.4.2 Proof That Lowenheim's Algorithm Produces a Most General Unifier

In this subsection we will prove that our Lowenheim builder gives a unifier that is most general; this will help us a lot in the top-level proof that the *Main_Lownheim* function gives an mgu.

Here is the subsection's resulting lemma. Given a unifiable term t, a unifier of t, then our Lowenheim builder (*build_lownheim_subst*) gives a most general unifier (mgu).

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

4.5 Proof of Correctness of Lowenheim_Main

In this section we prove that our own defined Lownheim function satisfies its two main requirements: 1) If a term is unifiable, then Lowenheim_Main function produces a most general unifier (mgu). 2) If a term is not unifiable, then Lownheim_Main function produces a None substitution. The final top-level proof is at the end of this section. To get there, we prove a series of intermediate lemmas that are needed for the final proof.

4.5.1 Utilities

In this section we provide helper "utility" lemmas and functions that are used in the proofs of intermediate lemmas that are in turn used in the final proof.

This is a function that converts a **option** `subst` to a `subst`. It is designed to be used mainly for **option** `substs` that are Some σ. If the input **option** `subst` is not Some and is None then it returns the nil substitution, but that case should not normally be considered. This function is useful because many functions and lemmas are defined for the substitution type not the option substitution type.

```
Definition convert_to_subst (so : option subst) : subst :=
  match so with
  | Some s ⇒ s
  | None ⇒ []
  end.
```

This is an intuitive helper lemma that proves that if an empty substitution is applied on any term t, then the resulting term is the same input term t.

```
Lemma empty_subst_on_term: ∀ (t : term),
  apply_subst t [] == t.
```

This another intutive helper lemma that states that if the empty substitution is applied on any term t, and the resulting term is equivalent to the ground term T0, then the input term t must be equivalent to the ground term T0.

```
Lemma app_subst_T0: ∀ (t : term),
 apply_subst t [] == T0 → t == T0.
```

This is another intutitve lemma that uses classical logic for it proof. It states that any term t, can be equivalent to the ground term $T0$ or it cannot be equivalent to it.

```
Lemma T0_or_not_T0: ∀ (t : term),
 t == T0 ∨ ¬ t == T0.
```

This is another intuitive helper lemma that states: if applying a substitution σ on a term t gives a term equivalent to T0 then there exists a substitution that applying it to term t gives a term equivalent to T0 (which is obvious since we already know σ exists for that task).

```
Lemma exists_subst: ∀ (t : term) (sig : subst),
  apply_subst t sig == T0 → ∃ s , apply_subst t s == T0.
```

```
Lemma t_id_eqv : ∀ (t : term),
  t == t.
```

This a helper lemma that states: if two *options* `substs` (specifically Some) are equal then the substitutions contained within the **option** `subst` are also equal.

```
Lemma eq_some_eq_subst (s1 s2: subst) :
  Some s1 = Some s2 → s1 = s2.
```

This a helper lemma that states: if the find_unifier function (the one that tries to find a ground unifier for term t) does not find a unifier (returns None) for an input term t then it not **True** (true not in "boolean format" but as a proposition) that the find_unifier function produces a Some `subst`. This lemma and the following ones that are similar, are very useful for the intermediate proofs because we are able to convert a proposition about the return type of the find_unifier function to an equivalent one, e.g. from None `subst` to Some `subst` and vice versa.

```
Lemma None_is_not_Some (t: term):
  find_unifier t = None →
  ∀ (sig: subst), ¬ find_unifier t = Some sig.
```

This a helper lemma similar to the previous one that states: if the find_unifier function (the one that tries to find a ground unifier for term t) finds a unifier (returns Some σ) for an input term t then it is not **True** (true not in "boolean format" but as a proposition) that the find_unifier function produces a None `subst`.

```
Lemma Some_is_not_None (sig: subst) (t: term):
  find_unifier t = Some sig → ¬ find_unifier t = None.
```

This a helper lemma similar to the previous ones that states: if the find_unifier function (the one that tries to find a ground unifier for term t) does not find a unifier that returns None for an input term t then it is **True** (true not in "boolean format" but as a proposition) that the find_unifier function produces a Some `subst`.

```
Lemma not_None_is_Some (t: term) :
  ¬ find_unifier t = None →
  ∃ sig : subst, find_unifier t = Some sig.
```

This is an intutitive helper lemma that uses classical logic to prove the validity of an alternate version of the contrapositive proposition: if p then q implies if not q then not p, but with each entity (proposition q and p) negated.

```
Lemma contrapositive_opposite : ∀ p q,
  (¬p → ¬q) →
  q → p.
```

This is an intutitive helper lemma that uses classical logic to prove the validity of the contrapositive proposition: if p then q implies not q then not p.

```
Lemma contrapositive : ∀ (p q : Prop),
  (p → q) →
  (¬q → ¬p).
```

4.5.2 Intermediate Lemmas

In this subsection we prove a series of lemmas for each of the two statements of the final proof, which were: 1) if a term is unifiable, then Lowenheim_Main function produces a most

general unifier (mgu). 2) if a term is not unifiable, then *Lownheim_Main* function produces a None substitution.

None Substitution Case

In this secton we prove intermediate lemmas useful for the second statement of the final proof: if a term is not unifiable, then *Lownheim_Main* function produces a None substitution.

Lemma to show that if find_unifier returns Some `subst`, the term is unifiable.

```
Lemma some_subst_unifiable: ∀ (t : term),
  (∃ sig , find_unifier t = Some sig) →
  unifiable t.
```

This lemma shows that if no substituion makes find_unifier to return Some `subst`, the it returns None.

```
Lemma not_Some_is_None (t: term) :
  ¬ (∃ sig , find_unifier t = Some sig) →
  find_unifier t = None.

Lemma not_unifiable_find_unifier_none_subst : ∀ (t : term),
  ¬ unifiable t → find_unifier t = None.
```

Some Substituion Case

In this secton we prove intermediate lemmas useful for the first statement of the final proof: if a term is unifiable, then Lowenheim_Main function produces a most general unifier (mgu).

Lemma to show that if find_unifier on an input term t returns Some σ, then σ is a unifier of t.

```
Lemma Some_subst_unifiable : ∀ (t : term) (sig : subst),
  find_unifier t = Some sig → unifier t sig.
```

This lemma shows that if there is a unifier, then there is a "ground unifier".

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

Lemma to show that if no substituion makes find_unifier return Some σ, then it returns None.

```
Lemma not_Some_not_unifiable (t: term) :
  (¬ ∃ sig , find_unifier t = Some sig) →
  ¬ unifiable t.
```

This lemma shows that if a term is unifiable then find_unifier returns Some σ.

```
Lemma unifiable_find_unifier_some_subst : ∀ (t : term),
  unifiable t →
```

(∃ *sig*, find_unifier t = Some *sig*).

This lemma shows that if a term is unifiable, then find_unifier returns a unifier.

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

4.5.3 Gluing Everything Together For the Final Proof

In this subsection we prove the two top-level final proof lemmas. Both of these proofs use the intermediate lemmas proved in the previous subsections.

The first one states that given a uniable term t and the fact that our Lowenheim builder produces an mgu, then the Lowenheim_Main function also produces an mgu.

```
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)).
```

This the final top-level lemma that encapsulates all our efforts so far. It proves the two main statements required for the final proof. The two statements, as phrased in the beginning of the chapter are: 1) if a term is unifiable, then our own defined Lowenheim_Main function produces a most general unifier (mgu). 2) if a term is not unifiable, then our own defined *Lownheim_Main* function produces a None substitution. The two propositions are related with the "∧" symbol (namely, the propositional "and") and each is proven seperately using the intermediate lemmas proven above. This is why the final top-level proof is relatively short, because a lot of the significant components of the proof have already been proven as intermediate lemmas.

```
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).
```

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 $x * x \approx_B x$, we can guarantee that there are no repeated variables in any given monomial. Similarly, because $x + x \approx_B 0$, 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} (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, lex compare l l = Eq.
```

Next, *antisymmetry*. This allows us to take a predicate or hypothesis about the comparison of two polynomials and reverse it.

For example, $l < m$ implies $m > l$.

```
Lemma lex_nat_antisym : ∀ l m,
  lex compare l m = CompOpp (lex compare m l).
```

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 : ∀ l m,
  lex compare l m = Eq ↔ l = m.

Lemma lex_neq : ∀ l m,
  lex compare l m = Lt ∨ lex compare l m = Gt ↔ l ≠ m.

Lemma lex_neq' : ∀ l m,
  (lex compare l m = Lt → l ≠ m) ∧
  (lex compare l m = Gt → l ≠ m).
```

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 compare returns that $l = m$, then this also means that $m = l$. More interesting is that if $l < m$, then $m > l$.

```
Lemma lex_rev_eq : ∀ l m,
  lex compare l m = Eq ↔ lex compare m l = Eq.

Lemma lex_rev_lt_gt : ∀ l m,
  lex compare l m = Lt ↔ lex compare m l = Gt.
```

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 change the lists' comparison.

```
Lemma lex_nat_cons : ∀ l m n,
  lex compare l m = lex compare (n :: l) (n :: m).

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 (a:A) l l',
  Permutation l l' →
  ¬ In a l →
  ¬ In a l'.
```

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 = [].
```

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:A) l m,
  incl l (a :: m) →
  ¬ In a l →
  incl l m.
```

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.
```

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

```
Lemma incl_Permutation : ∀ {A} (l l' m:list A),
  Permutation l l' →
  incl l m →
  incl l' m.
```

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

```
Lemma incl_nil : ∀ {X} (l:list X),
  incl l [] ↔ l = [].
```

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 (a:A) l m,
  incl (a :: l) m → In a m ∧ incl l m.
```

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 Aeq_dec (a:A) l l',
  Permutation l l' →
  count_occ Aeq_dec l a = count_occ Aeq_dec l' a.
```

The function count_occ also distributes over list concatenation, instead becoming addition. This is useful especially when dealing with count occurences of lists during induction.

```
Lemma count_occ_app : ∀ A (a:A) l m Aeq_dec,
  count_occ Aeq_dec (l ++ m) a =
  add (count_occ Aeq_dec l a) (count_occ Aeq_dec m a).
```

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) l,
  count_occ Aeq_dec (remove Aeq_dec a l) a = 0.

Lemma count_occ_neq_remove : ∀ {A} Aeq_dec (a b:A) l,
  a ≠ b →
  count_occ Aeq_dec (remove Aeq_dec a l) b =
  count_occ Aeq_dec l b.
```

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 flattening of each list are also permutations.

```
Lemma Permutation_concat : ∀ {A} (l m:list (list A)),
  Permutation l m →
  Permutation (concat l) (concat m).
```

Before the creation of this next lemma, it was relatively hard to reason about whether elements are in the flattening 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 flattened version.

```
Lemma In_concat_exists : ∀ A ll (a:A),
  (∃ l, In l ll ∧ In a l) ↔ In a (concat ll).
```

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

```
Lemma concat_map : ∀ {A B} (f:A→B) l,
  concat (map (fun a ⇒ [f a]) l) = map f l.
```

Another fact similar to the last is that if you flatten 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)).
```

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} (l:list A),
  concat (map (fun x ⇒ []) l) = (@nil A).
```

5.3.5 Facts about Forall and existsb

This is similar to the inverse of Forall; any element in a list l must hold for predicate p if Forall p is true of l.

```
Lemma Forall_In : ∀ A (l:list A) a p,
  In a l → Forall p l → p a.
```

In Coq, existsb is effectively the "or" to Forall's "and" when reasoning about lists. If there does not exist a single element in a list l where the predicate p holds, then $p\ a$ must be false for any element a of l.

```
Lemma existsb_false_forall : ∀ {A} p (l:list A),
  existsb p l = false →
  (∀ a, In a l → p a = false).
```

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 p (a:A) l,
  Forall p (a :: l) ↔ Forall p l ∧ p a.
```

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

```
Lemma Forall_remove : ∀ A Aeq_dec p (a:A) l,
  Forall p l → Forall p (remove Aeq_dec a l).
```

This next lemma is particularly useful for relating StronglySorted lists to Sorted lists; if some relation holds between all elements of a list, then this can be converted to the HdRel proposition used by Sorted.

```
Lemma Forall_HdRel : ∀ {X} r (x:X) l,
  Forall (r x) l → HdRel r x l.
```

Lastly, if some predicate p holds for all elements in a list l, and the elements of a second list m are all included in l, then p holds for all the elements in m.

```
Lemma Forall_incl : ∀ {X} p (l m:list X),
  Forall p l → incl m l → Forall p m.
```

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} Aeq_dec (a b:A) l,
  In a (remove Aeq_dec b l) → In a l.
```

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

Lemma StronglySorted_remove : $\forall$ {A} Aeq_dec r (a:A) l,
StronglySorted r l $\rightarrow$ StronglySorted r (remove Aeq_dec a l).

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 : $\forall$ A Aeq_dec (a:A) l,
$\neg$ In a l $\rightarrow$ remove Aeq_dec a l = l.

The function remove also distributes over list concatenation.

Lemma remove_distr_app : $\forall$ A Aeq_dec (a:A) l m,
remove Aeq_dec a (l ++ m) = remove Aeq_dec a l ++ remove Aeq_dec a m.

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 : $\forall$ A Aeq_dec (a:A) l l',
Permutation l l' $\rightarrow$
Permutation (remove Aeq_dec a l) (remove Aeq_dec a l').

The function remove is also associative with itself.

Lemma remove_remove : $\forall$ {A} Aeq_dec (a b:A) l,
remove Aeq_dec a (remove Aeq_dec b l) =
remove Aeq_dec b (remove Aeq_dec a l).

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

Lemma remove_pointless : $\forall$ {A Aeq_dec} (a:A) l m,
remove Aeq_dec a (remove Aeq_dec a l ++ m) =
remove Aeq_dec a (l ++ m).

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 the first two elements of that list must not be equal.

Lemma NoDup_neq : $\forall$ {A} l (a b:A),
NoDup (a :: b :: l) $\rightarrow$
$a \neq b$.

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 : $\forall$ A Aeq_dec (a:A) l,
NoDup l $\rightarrow$ NoDup (remove Aeq_dec a l).

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 (a:A) l,
  Forall (fun b ⇒ a ≠ b) l →
  NoDup l →
  NoDup (a :: l).
```

This lemma is really just a reformalization of NoDup_remove_2, which allows us to easily prove that some a 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} (a:A) l l1 l2,
  l = l1 ++ a :: l2 →
  NoDup l →
  ¬ In a l1 ∧ ¬ In a l2.
```

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 Aeq_dec (l:list A),
  NoDup l →
  nodup Aeq_dec l = l.
```

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

```
Lemma Sorted_nodup : ∀ A Aeq_dec r (l:list A),
  Relations_1.Transitive r →
  Sorted r l →
  Sorted r (nodup Aeq_dec l).
```

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 : ∀ l m,
  nodup Nat.eq_dec (l ++ nodup Nat.eq_dec m) = nodup Nat.eq_dec (l ++ m).
```

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).
```

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 {A} p l :
  fst (partition p l) = @filter A p l.
Lemma partition_filter_fst' : ∀ {A} p (l t f : list A),
    partition p l = (t, f) →
    t = @filter A p l .
```

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 {A:Type} := fun (p:A→bool) ⇒ fun a ⇒ negb (p a).
Lemma neg_true_false : ∀ {A} p (a:A),
  p a = true ↔ neg p a = false.
Lemma neg_false_true : ∀ {A} p (a:A),
  p a = false ↔ neg p a = true.
```

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 {A} p l :
  snd (partition p l) = @filter A (neg p) l.
Lemma partition_filter_snd' : ∀ {A} p (l t f : list A),
  partition p l = (t, f) →
  f = @filter A (neg p) l.
```

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 : ∀ A p (l t f : list A),
  partition p l = (t, f) →
  (∀ a, In a t → p a = true).
Lemma part_snd_false : ∀ A p (x t f : list A),
  partition p x = (t, f) →
  (∀ a, In a f → p a = false).
```

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

```
Lemma partition_Permutation : ∀ {A} p (l t f:list A),
  partition p l = (t, f) →
  Permutation l (t ++ f).
```

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 between 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 between all elements of the original list p, and therefore also holds it between the elements of the partitioned list.

```
Lemma part_Sorted : ∀ {X} (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.
```

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 ⇒ 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 : ∀ {A} (l:list (list A)),
```

```
  distribute [] l = [].

Lemma distribute_nil_r : ∀ {A} (l:list (list A)),
  distribute l [] = [].
```

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 Aeq_dec a (l:list A),
  In a (nodup_cancel Aeq_dec l) → In a l.

Lemma NoDup_nodup_cancel : ∀ A Aeq_dec (l:list A),
```

NoDup (nodup_cancel *Aeq_dec l*).

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 Aeq_dec (l:list A),
  NoDup l →
  nodup_cancel Aeq_dec l = l.

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

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).
```

The Permutation_nodup lemma 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 Aeq_dec (l l':list A),
  Permutation l l' →
  Permutation (nodup_cancel Aeq_dec l) (nodup_cancel Aeq_dec l').
```

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).
```

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*) = [].

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).
```

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).
```

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 an 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)).
```

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)).
```

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)).
```

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)).
```

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).
```

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.

Lemma parity_match_sym : ∀ {A Aeq_dec} (l m:list A),
  parity_match Aeq_dec l m ↔ parity_match Aeq_dec m l.

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.

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.
```

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.
```

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).
```

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.
```

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.
```

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).
```

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).
```

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.
```

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.
```

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).
```

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).
```

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)).
```

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 flattened 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.
```

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.
```

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.

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.
```

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 flattening.

```
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.
```

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.
```

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))).
```

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 $a \neq b$. 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).
```

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).
```

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.
```

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.
```

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.

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

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 [].

Lemma nil_is_poly :
  is_poly [].

Lemma one_is_poly :
  is_poly [[]].

Lemma var_is_poly : ∀ x,
  is_poly [[x]].
```

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 = [[]].
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 $y \leq x$, then 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).
End MonoOrder.

Lemma leb_both_eq : ∀ x y,
  is_true (MonoOrder.leb x y) →
  is_true (MonoOrder.leb y x) →
  x = y.
```

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.

Lemma VarOrder_Transitive :
  Relations_1.Transitive (fun x y ⇒ is_true (NatOrder.leb x y)).

Lemma mono_lt_Transitive : Relations_1.Transitive mono_lt.

Lemma MonoOrder_Transitive :
  Relations_1.Transitive (fun x y ⇒ is_true (MonoOrder.leb x y)).
```

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 why 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.
Lemma Permutation_VarSort_r : ∀ m n,
  Permutation m n ↔ Permutation m (VarSort.sort n).
Lemma Permutation_MonoSort_r : ∀ p q,
  Permutation p q ↔ Permutation p (sort q).
Lemma Permutation_MonoSort_l : ∀ p q,
  Permutation p q ↔ Permutation (sort p) q.
```

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.
```

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.

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 lemmas will come up a lot in future proofs, and has made some of our work much easier.

Lemma Permutation_sort_mono_eq : ∀ l m,
Permutation l m ↔ VarSort.sort l = VarSort.sort m.

Lemma Permutation_sort_eq : ∀ l m,
Permutation l m ↔ sort l = sort m.

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.

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.

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,
  Sorted (fun n m ⇒ is_true (leb n m)) m ∧ NoDup m →
  Sorted lt m.

Lemma MonoSort_Sorted : ∀ p,
  Sorted (fun n m ⇒ is_true (MonoOrder.leb n m)) p ∧ NoDup p →
  Sorted mono_lt p.
```

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.

Lemma Sorted_MonoSorted : ∀ (p : poly),
  Sorted mono_lt p →
  Sorted (fun n m ⇒ is_true (MonoOrder.leb n m)) p.
```

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_VarSorted : ∀ m,
  Sorted lt m → NoDup m.

Lemma NoDup_MonoSorted : ∀ p,
  Sorted mono_lt p → NoDup p.
```

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).

Lemma NoDup_VarSort : ∀ (m : mono),
  NoDup m → NoDup (VarSort.sort m).

Lemma NoDup_MonoSort : ∀ (p : poly),
  NoDup p → NoDup (MonoSort.sort p).
```

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 $x*x \approx_B x$, 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.
```

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 no duplicates, which is obvious considering that is exactly what make_mono does!

```
Lemma make_mono_is_mono : ∀ m,
  is_mono (make_mono m).
```

The proof for make_poly_is_poly is almost identical, with the addition of one part. The is_poly predicate 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).

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.
```

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).
```

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).
```

Due to the nature of sorting, make_mono is commutative across list concatenation.

```
Lemma make_mono_app_comm : ∀ m n,
  make_mono (m ++ n) = make_mono (n ++ m).
```

Finally, if 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.
```

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.
```

Because we have shown that sort and Permutation are equivalent, we can easily show that make_poly is commutative accross list concatenation.

```
Lemma make_poly_app_comm : ∀ p q,
  make_poly (p ++ q) = make_poly (q ++ p).
```

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).
```

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.
```

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 and remove, 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.
```

```
Lemma no_sort_MonoSorted : ∀ p,
  Sorted mono_lt p →
  MonoSort.sort p = p.
```

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).
```

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.
```

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).
```

Similarly, if we already know that all of the elements in a list are monomials, 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.
```

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).
```

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.
```

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

```
Lemma no_make_poly : ∀ p,
  is_poly p →
  make_poly p = p.
```

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).

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 of 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).
```

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).
```

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 polynomial.

```
Definition addPP (p q : poly) : poly :=
  make_poly (p ++ q).

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

Lemma addPP_is_poly : ∀ p q,
  is_poly (addPP p q).
```

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).
```

```
Hint Resolve addPP_is_poly mulPP_is_poly.
```

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.

Lemma mulPP_mulPP' : ∀ (p q : poly),
  mulPP p q = mulPP' p q.
```

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.

Lemma mulPP'_mulPP'' : ∀ p q,
  mulPP' p q = mulPP'' p q.
```

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.
```

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).
Lemma mulPP''_mulPP''' : ∀ p q,
  mulPP'' p q = mulPP''' p q.
```

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.
Lemma mulPP_mulPP''' : ∀ p q,
  mulPP p q = mulPP''' p q.
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 identity for each addition and multiplication. First is the additive inverse, which states that forall terms x, $(x + x) \downarrow_P 0$.

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 = [].
```

6.6.2 Axiom 2: Additive Identity

Next, we prove the additive identity: for all terms x, $(0 + x) \downarrow_P = x \downarrow_P$. 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.

Lemma addPP_0r : ∀ p,
  is_poly p →
  addPP p [] = p.
```

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, $(x{*}1) \downarrow_P = x \downarrow_P$.

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

```
Lemma mulPP_1r : ∀ p,
  is_poly p →
  mulPP p [[]] = p.
```

6.6.4 Axiom 4: Multiplicative Inverse

Next is the multiplicative inverse, which states that for any term x, $(0 * x) \downarrow_P = 0$.

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

```
Lemma mulPP_0 : ∀ p,
  mulPP [] p = [].

Lemma mulPP_0r : ∀ p,
  mulPP p [] = [].
```

6.6.5 Axiom 5: Commutativity of Addition

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

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.
```

6.6.6 Axiom 6: Associativity of Addition

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

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 ++ q ++ r is a permutation of q ++ r ++ p.

```
Lemma addPP_assoc : ∀ p q r,
  addPP (addPP p q) r = addPP p (addPP q r).
```

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, $(x * y) \downarrow_P = (y * x) \downarrow_P$. 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 simplify 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.
```

6.6.8 Axiom 8: Associativity of Multiplication

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

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).
```

Next is a deceptively easy lemma map_app_make_poly, which 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).
```

The map_app_make_poly lemma 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.
```

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

```
Lemma mulMP'_app : ∀ p q m,
  mulMP' (p ++ q) m =
  mulMP' p m ++ mulMP' q m.
```

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.
```

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. Thus 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; $((q*[a :: p]) * m)\downarrow_P$ becomes $((q*a)*m)\downarrow_P ++ ((q*p)*m)\downarrow_P$. At the end of all of this rewriting, we are left with the original $(p*q*r)\downarrow_P$ as the last term of both sides, and $(q*p*m)\downarrow_P$ and $(q*r*a)\downarrow_P$ 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 $((q * a) * m) \downarrow_P$, and on the right we have $((q * m) * a) \downarrow_P$. 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).
```

6.6.9 Axiom 9: Multiplicative Identity - Self

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

To begin, we prove that this holds for monomials; $(m * m) \downarrow_P = m \downarrow_P$. 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 as. After that, perm_skip and the induction hypothesis solve the lemma.

```
Lemma make_mono_self : ∀ m,
  is_mono m →
  make_mono (m ++ m) = m.
```

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 $a * a$ 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 $(a * [a :: p]) \downarrow_P ++ ([a :: p] * p) \downarrow_P = p \downarrow_P$ 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 $(a * [a :: p]) \downarrow_P ++ (a * [a :: p]) \downarrow_P ++ (p * p) \downarrow_P = p \downarrow_P$ which is much more workable! We know that $(p * p) \downarrow_P = p \downarrow_P$ 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.
```

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, $(x * (y + z)) \downarrow_P = ((x * y) + (x * z)) \downarrow_P$.

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).
```

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 $((p + q) * a) \downarrow_P ++ ((p + q) * r) \downarrow_P$.

We then apply similar tactics on the right side, to convert our goal to a form similar to $(p * a + q * a + p * r + q * r) \downarrow_P$. The two terms containing r are easy to deal with, since we know they are equal to the $((p + q) * r) \downarrow_P$ we have on the left side due to the induction hypothesis. Similarly, the first two terms are known to be equal to $((p + q) * a) \downarrow_P$ 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).
```

For convenience, we also prove that distribution can be applied from the 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).
```

6.7 Other Facts About Polynomials

Now that we have proven the core ten axioms proven, there are a few more useful lemmas that we will prove to assist us in future parts of the development.

6.7.1 More Arithmetic

Occasionally, when dealing with multiplication, we already know that one of the variables being multiplied in is less than the rest, meaning it would end up at the front of the list after sorting. For convenience and to bypass the work of dealing with the calls to sort and nodup_cancel, the below lemma allows us to rewrite with this concept.

```
Lemma mulPP_mono_cons : ∀ x m,
  is_mono (x :: m) →
  mulPP [[x]] [m] = [x :: m].
```

Similarly, if we already know some monomial is less than the polynomials it is being added to, then the monomial will clearly end up at the front of the list.

```
Lemma addPP_poly_cons : ∀ m p,
  is_poly (m :: p) →
  addPP [m] p = m :: p.
```

An interesting arithmetic fact is that if we multiply the term $((p*q)+r)\downarrow_P$ by $(1+q)\downarrow_P$, we effectively eliminate the $(p*q)\downarrow_P$ term and are left with $((1+q)*r)\downarrow_P$. This will come into play later in the development, as we look to begin building unifiers.

```
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.
```

6.7.2 Reasoning about Variables

To more easily deal with the vars definition, we have defined a few definitions about it. First, if some x is in the variables of make_poly p, then it must have been in the vars of p originally. Note that this is not true in the other direction, as nodup_cancel may remove some variables.

```
Lemma make_poly_rem_vars : ∀ p x,
  In x (vars (make_poly p)) →
  In x (vars p).
```

An interesting observation about addPP and our vars function is that clearly, the variables of some $(p+q)\downarrow_P$ is a subset of the variables of p combined with the variables of q. The next lemma is a more convenient formulation of that fact, using a list of variables xs rather than comparing them directly.

```
Lemma incl_vars_addPP : ∀ p q xs,
  incl (vars p) xs ∧ incl (vars q) xs →
  incl (vars (addPP p q)) xs.
```

We would like to be able to prove a similar fact about mulPP, but before we can do so, we need to know more about the distribute function. This lemma states that if some a is in the variables of distribute l m, then it must have been in either vars l or vars m originally.

```
Lemma In_distribute : ∀ (l m:poly) a,
  In a (vars (distribute l m)) →
  In a (vars l) ∨ In a (vars m).
```

We can then use this fact to prove our desired fact about mulPP; the variables of $(p*q)\downarrow_P$ are a subset of the variables of p and the variables of q. Once again, this is formalized in a way that is more convenient in later proofs, with an extra list xs.

```
Lemma incl_vars_mulPP : ∀ p q xs,
  incl (vars p) xs ∧ incl (vars q) xs →
  incl (vars (mulPP p q)) xs.
```

6.7.3 Partition with Polynomials

When it comes to actually performing successive variable elimination later in the development, the partition function will play a big role, so we have opted to prove a few useful facts about its relation to polynomials now.

First is that if you separate a polynomial with any function *f*, you can get the original polynomial back by adding together the two lists returned by partition. This is relatively easy to prove thanks to the lemma partition_Permutation we proved during list_util.

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

In addition, if you partition some polynomial *p* with any function *f*, the resulting two lists will both be proper polynomials, since partition does not affect the order.

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

6.7.4 Multiplication and Remove

Lastly are some rather complex lemmas relating remove and multiplication. Similarly to the partition lemmas, these will come to play a large roll in performing successive variable elimination later in the development.

First is an interesting fact about removing from monomials. If there are two monomials which are equal after removing some *x*, and either both contain *x* or both do not contain *x*, then they must have been equal originally. This proof begins by performing double induction, and quickly solving the first three cases.

The fourth case is rather long, and begins by comparing if the *a* and *a0* at the head of each list are equal. The case where they are equal is relatively straightforward; we must also destruct if $x = a = a0$, but regardless of whether they are equal or not, we can easily prove this with the use of the induction hypothesis.

The case where $a \neq a0$ should be a contradiction, as that element is at the head of both lists, and we know the lists are equal after removing *x*. We begin by destructing whether or not *x* is in the two lists. In the case where it is not in either, we can quickly solve this, as we know the call to remove will do nothing, which immediately gives us the contradiction.

In the case where *x* is in both, we begin by using in_split to rewrite both lists to contain *x*. We then use the fact that there are no duplicates in either list to show that *x* is not in *l1*, *l2*, *l1'*, or *l2'*, and therefore the calls to remove will do nothing. This leaves us with a hypothesis that $l1 ++ l2 = l1' ++ l2'$. To finish the proof, we destruct *l1* and *l1'* to further compare the head of each list.

In the case where they are both empty, we arrive at a contradiction immediately, as this implies the head of both lists is x and therefore contradicts that $a \neq a0$. In the case where they are both lists, doing inversion on our remove hypothesis gives us that the head of each list is equal again, also contradicting that $a \neq a0$.

In the other two cases, we rewrite with the in_split hypotheses into the is_mono hypotheses. In both cases, we result in one statement that a comes before $a0$ in the monomial, and one statement that $a0$ comes before a in the monomial. With the help of **StronglySorted**, we are able to turn these into $a < a0$ and $a0 < a$, which contradict each other to finish the proof.

```
Lemma remove_Sorted_eq : ∀ 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'.
```

Next is that if we map remove across a polynomial where every monomial contains x, there will still be no duplicates at the end.

```
Lemma NoDup_map_remove : ∀ x p,
  is_poly p →
  (∀ m, In m p → In x m) →
  NoDup (map (remove var_eq_dec x) p).
```

Building off that, if every monomial in a list *does not* contain some x, then appending x to every monomial and calling make_mono still will not create any duplicates.

```
Lemma NoDup_map_app : ∀ x l,
  is_poly l →
  (∀ m, In m l → ¬ In x m) →
  NoDup (map make_mono (map (fun a ⇒ a ++ [x]) l)).
```

This next lemma is relatively straightforward, and really just served to remove the calls to sort and nodup_cancel for convenience when simplifying a mulPP.

```
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)).
```

Building off of the previous lemma, this one serves to remove the calls to make_poly entirely, and instead replace mulPP with just the map app. We can do this because we know that x is not in any of the monomials, so nodup_cancel will have no effect as we proved earlier.

```
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').
```

Finally, we combine the lemmas in this section to prove that, if there is some polynomial p that has x in every monomial, removing and then re-appending x to every monomial results in a list that is a permutation of the original polynomial.

```
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)).
```

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 lists 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.
```

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).

Lemma substM_is_poly : ∀ s m,
  is_poly (substM s m).

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

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.
```

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.
```

```
Lemma substP_cons : ∀ x p,
  (∀ m, In m p → ¬ In x m) →
  ∀ q s, substP ((x, q) :: s) p = substP s p.
```

Substitutions applied to constants have no effect.

```
Lemma substP_1 : ∀ s,
  substP s [[]] = [[]].
```

```
Lemma substP_0 : ∀ s,
  substP s [] = [].
```

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

```
Lemma empty_substM : ∀ m,
  is_mono m →
  substM [] m = [m].
```

```
Lemma empty_substP : ∀ p,
  is_poly p →
  substP [] p = p.
```

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).
```

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.

Lemma substM_Permutation : ∀ s m n,
  Permutation m n →
  Permutation (substM s m) (substM s n).
```

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.
```

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)))))).

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)))))).
```

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).
```

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.
```

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.
```

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 [] [].
```

```
Lemma empty_reprod_unif : reprod_unif [] [].
```

```
Lemma empty_mgu : mgu [] [].
```

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; $1 \overset{?}{\approx}_B 0$ or $0 \overset{?}{\approx}_B 0$. 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).
```

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).

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).
```

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)).
```

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)).
```

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).
```

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.
```

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)).
```

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.
```

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.
```

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)).
```

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).
```

```
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).
```

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.
```

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 : ∀ 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.
```

```
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 σ is a reproductive unifier of $p' \stackrel{?}{\approx}_B 0$, then we can build a substitution σ' which is a reproductive unifier of $p \stackrel{?}{\approx}_B 0$. The way σ' is built from σ is defined in build_subst. Another replacement is added to σ of the form $\{x \mapsto (x*(\sigma(q)+1)+\sigma(r)) \downarrow_P\}$ to construct σ'.

```
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).
```

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

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

```
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.
```

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

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 σ 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 σ, then the range of σ 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.
```

If sveVars produces a substitution σ for the polynomial p, then σ 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.
```

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.
```

```
Hint Resolve NoDup_vars incl_refl.
```

If sveVars produces a substitution σ for the polynomial p, then σ 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.
```

If sve produces a substitution σ for the polynomial p, then σ 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.
```