Contents

Chapter 1

Library B_Unification.intro

1.1 Introduction

1.2 Unification

Before defining what unification is, there is some terminology to understand. A *term* is either a variable or a function applied to terms. 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 examples 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. 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. It is common for substitutions to be referred to as mappings from terms to terms. A substitution s can be extended to this form by defining $s'(u)$ for two cases of u. If u is a variable, then $s'(u) = s(u)$. If u is a function $f(u1, ..., un)$, then $s'(u) = f(s'(u1), ..., s'(un))$.

Unification is the process of solving a set of equations between two terms. The set of equations is referred to as a unification problem. 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. If two terms are equal with respect to an equational theory, then the unification is semantic.

The goal of unification is to solve equations, which means to produce a substitution that unifies those equations. A substitution s *unifies* an equation $u =? v$ if applying s to both sides makes them equal $s(u) = s(v)$. In this case, we call s a *solution* or *unifier*.

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 s is more general than *s'* if there exists a third substitution $\mathtt{t}$ such that $s'(u) = \mathtt{t}(s(u))$ for any term u.

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 $x \times y$ and $y \times x$ are not syntactically equal, but would be equal modulo commutativity of multiplication. (more about solving these problems / why simpler...)

1.2.2 Semantic Unification

This kind of unification involves an equational theory. Given a set of identities E, we write that two terms u and v are equal with regards to E as $u =E\ v$. This means that identities of E can be applied to u as *u'* and v as *v'* in some way to make them syntactically equal, $u' = v'$. As an example, let C be the set $\{f(x, y) = f(y, x)\}$. This theory C axiomatizes the commutativity of the function *f*. It would then make sense to write $f(a, x) =C\ f(x, a)$. In general, for an arbitrary E, the problem of E-unification is undecidable.

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 $\times$. The set of identities B is defined as the set $\{x + y = y + x, (x + y) + z = x + (y + z), x + x = 0, 0 + x = x, x \times (y + z) = (x \times y) + (x \times z), x \times y = y \times x, (x \times y) \times z = x \times (y \times z), x \times x = x, 0 \times x = 0, 1 \times x = x\}$. This set is equivalent to the theory of real numbers with the addition of $x + x = 0$ and $x \times x = x$.

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 $\mathtt{t} = 0$. Given a problem in its normal form $\{s1 = t1, ..., sn = t2\}$, we can transform it into $\{s1 + t1 = 0, ..., sn + tn = 0\}$ using a simple fact. The equation $s = \mathtt{t}$ is equivalent to $s + \mathtt{t} = 0$ since adding $\mathtt{t}$ to both sides of the equation turns the right hand side into $\mathtt{t} + \mathtt{t}$ which simplifies to 0. Then, given a problem $\{t1 = 0, ..., tn = 0\}$, we can transform it into $\{(t1 + 1) \times ... \times (tn + 1) = 1\}$. Unifying both of these sets is equivalent because if any t1, ..., tn is 1 the problem is not unifiable. Otherwise, if every t1, ..., tn can be made to equal 0, then both problems will be solved.

1.3 Formal Verification

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

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

1.3.1 Proof Assistance

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

1.3.2 Verifying Systems

1.3.3 Verifying Theories

1.4 Importance

1.5 Development

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

We chose to implement two data structures for representing the terms of a Boolean unification problem, and two algorithms for performing unification. The two data structures chosen are an inductive Term type and lists of lists representing polynomial-form terms. The two algorithms are Lowenheim's 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 – first as a "Term" inductive type, and then as lists of lists representing terms in polynomial form.

The Term inductive type, used in the proof of Lowenheim's algorithm, is very simple and rather intuitive – 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's algorithm and successive variable elimination.

The first solution, Lowenheim's algorithm, is built on top of the term inductive type. Lowenheim's 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's algorithm is implemented in the file *lowenheim.v*, and the proof of correctness is in *lowenheim_proof.v*.

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

Chapter 2

Library B_Unification.terms

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

2.1 Introduction

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

2.2 Terms

2.2.1 Definitions

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

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

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

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

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

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

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

Notation "x + y" := (SUM x y) (at level 50, left associativity).
Notation "x * y" := (PRODUCT x y) (at level 40, left associativity).
```

2.2.2 Axioms

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

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

```
Parameter eqv : term → term → Prop.
Infix " == " := eqv (at level 70).

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

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

Axiom sum_id : ∀ x, T0 + x == x.

Axiom sum_x_x : ∀ x, x + x == T0.

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

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

Axiom mul_x_x : ∀ x, x × x == x.

Axiom mul_T0_x : ∀ x, T0 × x == T0.

Axiom mul_id : ∀ x, T1 × x == x.

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

Axiom term_sum_symmetric :
  ∀ x y z, x == y ↔ x + z == y + z.
```

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

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

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.
Hint Resolve x_equal_y_x_plus_y.
```

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

```
Lemma sum_id_sym :
  ∀ x, x + T0 == x.

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

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

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

Definition subst := list replacement.

Implicit Type s : subst.
```

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

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

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

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

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

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

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

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

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

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.

```
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 :
  $\forall$ s x y, apply_subst x s + apply_subst y s == apply_subst (x + y) s.

Lemma subst_associative :
  $\forall$ s x y, apply_subst x s $\times$ apply_subst y s == apply_subst (x $\times$ y) s.

Lemma subst_sum_distr_opp :
  $\forall$ s x y, apply_subst (x + y) s == apply_subst x s + apply_subst y s.

Lemma subst_mul_distr_opp :
  $\forall$ s x y, apply_subst (x $\times$ y) s == apply_subst x s $\times$ apply_subst y s.

Lemma var_subst:
  $\forall$ (v : var) (ts : **term**) ,
  (apply_subst (VAR v) (cons (v , ts) nil)) == ts.

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

Lemma id_subst:
  $\forall$ (t : **term**) (l : var_set),
  apply_subst t (build_id_subst l) == t.

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 $\times$ 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) :: nil).
```

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) :: nil)).
```

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).
Lemma unifier_subset_imply_superset :
  ∀ s t r, unifier t s → unifier t (r :: 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

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

Definition more_general_substitution (s s': subst) (t : term) : Prop :=
  ∃ delta, substitution_composition s s' delta t.
Definition most_general_unifier (t : term) (s : subst) : Prop :=
  (unifier t s) → (∀ (s' : subst), unifier t s' → more_general_substitution s s' t ).
Definition reproductive_unifier (t : term) (sig : subst) : Prop :=
  unifier t sig →
  ∀ (tau : subst) (x : var),
  unifier t tau →
  (apply_subst (apply_subst (VAR x) sig ) tau) == (apply_subst (VAR x) tau).
Lemma reproductive_is_mgu : ∀ (t : term) (u : subst),
  reproductive_unifier t u →
  most_general_unifier t u.
```

2.8 Auxilliary Computational Operations and Simplifications

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

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

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

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

Chapter 3

Library B_Unification.lowenheim_formula

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

Fixpoint build_on_list_of_vars (list_var : var_set) (s : term) (sig1 : subst) (sig2 : subst) :
subst :=
  match list_var with
   | nil ⇒ nil
   | v' :: v ⇒
      (cons (v' , (s + T1) × (apply_subst (VAR v') sig1 ) + s × (apply_subst (VAR v' )
sig2 ) )
            (build_on_list_of_vars v s sig1 sig2) )
  end.
Definition build_lowenheim_subst (t : term) (tau : subst) : subst :=
  build_on_list_of_vars (term_unique_vars t) t (build_id_subst (term_unique_vars t)) tau.
```

2.2 Lowenheim's algorithm

```
Definition update_term (t : term) (s' : subst) : term :=
  (simplify (apply_subst t s' ) ).
Definition term_is_T0 (t : term) : bool :=
  (identical t T0).
Inductive subst_option: Type :=
     | Some_subst : subst → subst_option
     | None_subst : subst_option.
Fixpoint rec_subst (t : term) (vars : var_set) (s : subst) : subst :=
  match vars with
    | nil ⇒ s
    | v' :: v ⇒
        if (term_is_T0
              (update_term (update_term t (cons (v' , T0) s) )
                            (rec_subst (update_term t (cons (v' , T0) s) )
                                      v (cons (v' , T0) s)) )
                           )
          then
               (rec_subst (update_term t (cons (v' , T0) s) )
                                       v (cons (v' , T0) s))
         else
          if (term_is_T0
               (update_term (update_term t (cons (v' , T1) s) )
                             (rec_subst (update_term t (cons (v' , T1) s) )
                                       v (cons (v' , T1) s)) ) )
          then
               (rec_subst (update_term t (cons (v' , T1) s) )
                                       v (cons (v' , T1) s))
          else
               (rec_subst (update_term t (cons (v' , T0) s) )
                                       v (cons (v' , T0) s))
     end.

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

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

2.3 Lowenheim testing

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

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

Require Export lowenheim_formula.
```

3.1 Declarations and their lemmas useful for the proof

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

Lemma sub_term_id :
  ∀ (t : term),
  sub_term t t.

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

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

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

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

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

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

Lemma elt_not_in_list:
 ∀ (x: var) (a : var) (l : list var),
   ¬ (In x (a::l)) →
   x ≠ a ∧ ¬ (In x l).

Lemma in_list_of_var_term_of_var:
∀ (x : var),
   In x (term_unique_vars (VAR x)).

Lemma var_in_out_list:
   ∀ (x : var) (lvar : list var),
   (In x lvar) ∨ ¬ (In x lvar).
```

3.2 Proof that Lownheim's algorithm unifes a given term

```
Lemma helper1_easy:
 ∀ (x: var) (lvar : list var) (sig1 sig2 : subst) (s : term),
 (In x lvar) →
   apply_subst (VAR x) (build_on_list_of_vars lvar s sig1 sig2)
   ==
   apply_subst (VAR x) (build_on_list_of_vars (cons x nil) s sig1 sig2).

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

Lemma subs_distr_vars_ver2 :
   ∀ (t t' : term) (s : term) (sig1 sig2 : subst),
   (sub_term t t') →
   apply_subst t (build_on_list_of_vars (term_unique_vars t') s sig1 sig2)
     ==
   (s + T1) × (apply_subst t sig1) + s × (apply_subst t sig2).
```

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 .

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

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

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

Lemma lowenheim_rephrase1_easy :
  ∀ (l : **list** var) (x : var) ($sig1$: subst) ($sig2$: subst) (s : **term**),
  (In x l) →
  (apply_subst (VAR x) (build_on_list_of_vars l s $sig1$ $sig2$)) ==
  (s + T1) × (apply_subst (VAR x) $sig1$) + s × (apply_subst (VAR x) $sig2$).

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

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).

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

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).

Lemma lowenheim_reproductive:
  ∀ (t : **term**) (tau : subst),

```
  (unifier t tau) →
  reproductive_unifier t (build_lowenheim_subst t tau) .
```

3.3.b lowenheim builder gives a most general unifier

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

3.4 extension to include Main function and subst_option

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

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

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

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)) .

Lemma lowenheim_main_most_general_unifier:
 ∀ (t: term),
 (unifiable t) → most_general_unifier t (convert_to_subst (Lowenheim_Main t)).
```

Chapter 5

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 terms.
```

5.1 Introduction

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

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

5.2 Monomials and Polynomials

5.2.1 Data Type Definitions

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

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

A polynomial, then, is a list of monomials.

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

5.2.2 Comparisons of monomials and polynomials

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

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

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

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

```
Theorem lex_nat_refl : ∀ (l : list nat), lex compare l l = Eq.
```

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

```
Theorem lex_nat_antisym : ∀ (l1 l2 : list nat),
  lex compare l1 l2 = CompOpp (lex compare l2 l1).

Lemma lex_eq : ∀ n m,
  lex compare n m = Eq ↔ n = m.

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

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

Lemma lex_rev_eq : ∀ n m,
  lex compare n m = Eq ↔ lex compare m n = Eq.
Lemma lex_rev_lt_gt : ∀ n m,
  lex compare n m = Lt ↔ lex compare m n = 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 chance the lists' comparison.

```
Theorem lex_nat_cons : ∀ (l1 l2 : list nat) n,
  lex compare l1 l2 = lex compare (n::l1) (n::l2).
Hint Resolve lex_nat_refl lex_nat_antisym lex_nat_cons.
```

5.2.3 Stronger Definitions

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

Monomials are simply sorted lists of natural numbers.

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

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

```
Definition is_poly (p : poly) : Prop :=
  Sorted (fun m n ⇒ lex compare m n = Lt) p ∧ ∀ m, In m p → is_mono m.
Hint Unfold is_mono is_poly.
Hint Resolve NoDup_cons NoDup_nil Sorted_cons.
Definition vars (p : poly) : list var :=
  nodup var_eq_dec (concat p).

Lemma NoDup_vars : ∀ (p : poly),
  NoDup (vars p).
Lemma no_vars_is_ground : ∀ p,
  is_poly p →
  vars p = [] →
  p = [] ∨ p = [[]].
Lemma in_mono_in_vars : ∀ x p,
  (∀ m : mono, In m p → ¬ In x m) ↔ ¬ In x (vars p).
```

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

```
Lemma mono_order : ∀ x y m,
  is_mono (x :: y :: m) →
  x < y.
```

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.

```
Lemma poly_order : ∀ m n p,
  is_poly (m :: n :: p) →
  lex compare m n = Lt.
```

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

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

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

```
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]].

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

5.3 Functions over Monomials and Polynomials

```
Module Import VarSort := NatSort.

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.
```

Lemma In_remove : ∀ {A:Type} Aeq_dec a b (l:list A),
  In a (remove Aeq_dec b l) → In a l.

Lemma StronglySorted_remove : ∀ {A:Type} Aeq_dec Rel a (l:list A),
  StronglySorted Rel l → StronglySorted Rel (remove Aeq_dec a l).

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

Lemma remove_Sorted_eq : ∀ (A:Type) Aeq_dec x Rel (l l':list A),
  NoDup l →
  NoDup l' →
  Sorted Rel l →
  Sorted Rel l' →
  remove Aeq_dec x l = remove Aeq_dec x l' →
  l = l'.

Lemma remove_distr_app : ∀ (A:Type) Aeq_dec x (l l':list A),
  remove Aeq_dec x (l ++ l') = remove Aeq_dec x l ++ remove Aeq_dec x l'.

Lemma nodup_cancel_in : ∀ (A:Type) Aeq_dec a (l:list A),
  In a (nodup_cancel Aeq_dec l) → In a l.

Lemma NoDup_remove : ∀ (A:Type) Aeq_dec a (l:list A),
  NoDup l → NoDup (remove Aeq_dec a l).

Lemma NoDup_nodup_cancel : ∀ (A:Type) Aeq_dec (l:list A),
NoDup (nodup_cancel Aeq_dec l).

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

Lemma no_nodup_NoDup : ∀ (A:Type) Aeq_dec (l:list A),
  NoDup l →
  nodup Aeq_dec l = l.

Lemma no_nodup_cancel_NoDup : ∀ (A:Type) Aeq_dec (l:list A),
  NoDup l →
  nodup_cancel Aeq_dec l = l.

Lemma count_occ_Permutation : ∀ (A:Type) Aeq_dec a (l l':list A),
  Permutation l l' →
  count_occ Aeq_dec l a = count_occ Aeq_dec l' a.

Lemma Permutation_not_In : ∀ (A:Type) a (l l':list A),
  Permutation l l' →
  ¬ In a l →
  ¬ In a l'.

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

```
  Permutation l l' →
  Permutation (nodup_cancel Aeq_dec l) (nodup_cancel Aeq_dec l').
Require Import Orders.
Module MonoOrder <: TotalLeBool.
  Definition t := mono.
  Definition leb x y :=
    match lex compare x y with
    | Lt ⇒ true
    | Eq ⇒ true
    | Gt ⇒ false
    end.
  Infix "<=m" := leb (at level 35).
  Theorem leb_total : ∀ a1 a2, (a1 ≤m a2 = true) ∨ (a2 ≤m a1 = true).
End MonoOrder.
Module Import MonoSort := Sort MonoOrder.
Lemma VarOrder_Transitive :
  Relations_1.Transitive (fun x y : nat ⇒ is_true (NatOrder.leb x y)).
Lemma MonoOrder_Transitive :
  Relations_1.Transitive (fun x y : list nat ⇒ is_true (MonoOrder.leb x y)).
Lemma NoDup_neq : ∀ {X:Type} (m : list X) a b,
  NoDup (a :: b :: m) →
  a ≠ b.
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 VarSort_Sorted : ∀ (m : mono),
  Sorted (fun n m ⇒ is_true (leb n m)) m ∧ NoDup m → Sorted lt m.
Lemma Sorted_VarSorted : ∀ (m : mono),
  Sorted lt m →
  Sorted (fun n m ⇒ is_true (leb n m)) m.
Lemma In_sorted : ∀ a l,
  In a l ↔ In a (sort l).
Lemma HdRel_mono_le_lt : ∀ a p,
  HdRel (fun n m ⇒ is_true (MonoOrder.leb n m)) a p ∧ NoDup (a::p) →
  HdRel (fun n m ⇒ lex compare n m = Lt) a p.
Lemma MonoSort_Sorted : ∀ (p : poly),
  Sorted (fun n m ⇒ is_true (MonoOrder.leb n m)) p ∧ NoDup p →
  Sorted (fun n m ⇒ lex compare n m = Lt) p.
Lemma Sorted_MonoSorted : ∀ (p : poly),
  Sorted (fun n m ⇒ lex compare n m = Lt) p →
  Sorted (fun n m ⇒ is_true (MonoOrder.leb n m)) p.
Lemma NoDup_MonoSorted : ∀ (p : poly),
  Sorted (fun n m ⇒ lex compare n m = Lt) p →
  NoDup p.
Lemma NoDup_VarSorted : ∀ (m : mono),
  Sorted lt m → NoDup m.
Lemma NoDup_VarSort : ∀ (m : mono),
  NoDup m → NoDup (VarSort.sort m).
Lemma NoDup_MonoSort : ∀ (p : poly),
  NoDup p → NoDup (MonoSort.sort p).
Definition make_mono (l : list nat) : mono :=
  VarSort.sort (nodup var_eq_dec l).
Definition make_poly (l : list mono) : poly :=
  MonoSort.sort (nodup_cancel mono_eq_dec (map make_mono l)).
Lemma make_mono_is_mono : ∀ m,
  is_mono (make_mono m).
Lemma make_poly_is_poly : ∀ p,
  is_poly (make_poly p).
Lemma make_mono_In : ∀ x m,
  In x (make_mono m) → In x m.
Lemma no_make_mono : ∀ m,
  is_mono m →
  make_mono m = m.
Lemma remove_is_mono : ∀ x m,
  is_mono m →
  is_mono (remove var_eq_dec x m).
Lemma no_map_make_mono : ∀ p,
  (∀ m, In m p → is_mono m) →
  map make_mono p = p.
Lemma unsorted_poly : ∀ p,
  NoDup p →
  (∀ m, In m p → is_mono m) →
  nodup_cancel mono_eq_dec (map make_mono p) = p.
Definition addPP (p q : poly) : poly :=
  make_poly (p ++ q).
Definition distribute {A} (l m : list (list A)) : list (list A) :=
  concat (map (fun a:(list A) ⇒ (map (app a) l)) m).
Definition mulPP (p q : poly) : poly :=
  make_poly (distribute p q).
Lemma addPP_is_poly : ∀ p q,
  is_poly (addPP p q).
Lemma leb_both_eq : ∀ x y,
  is_true (MonoOrder.leb x y) →
  is_true (MonoOrder.leb y x) →
  x = y.
Lemma Permutation_incl : ∀ {A} (l m : list A),
  Permutation l m → incl l m ∧ incl m l.
Lemma incl_cons_inv : ∀ (A:Type) (a:A) (l m : list A),
  incl (a :: l) m → In a m ∧ incl l m.
Lemma Forall_In : ∀ (A:Type) (l:list A) a Rel,
  In a l → Forall Rel l → Rel a.
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.
Lemma Permutation_sort_mono_eq : ∀ (l m:mono),
  Permutation l m ↔ VarSort.sort l = VarSort.sort m.
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.
Lemma Permutation_sort_eq : ∀ l m,
  Permutation l m ↔ sort l = sort m.
Lemma sort_app_comm : ∀ l m,
  sort (l ++ m) = sort (m ++ l).
Lemma sort_nodup_cancel_assoc : ∀ l,
  sort (nodup_cancel mono_eq_dec l) = nodup_cancel mono_eq_dec (sort l).
Lemma addPP_comm : ∀ p q,
  addPP p q = addPP q p.
Hint Unfold addPP mulPP.
Lemma mulPP_l_r : ∀ p q r,
  p = q →
  mulPP p r = mulPP q r.
Lemma mulPP_0 : ∀ p,
  mulPP [] p = [].
Lemma addPP_0 : ∀ p,
  is_poly p →
  addPP [] p = p.
Lemma addPP_0r : ∀ p,
  is_poly p →
  addPP p [] = p.
Lemma addPP_p_p : ∀ p,
  addPP p p = [].
Lemma addPP_assoc : ∀ p q r,
  addPP (addPP p q) r = addPP p (addPP q r).
Lemma mulPP_1r : ∀ p,
  is_poly p →
  mulPP p [[]] = p.
Lemma mulPP_assoc : ∀ p q r,
  mulPP (mulPP p q) r = mulPP p (mulPP q r).
Lemma mulPP_comm : ∀ p q,
  mulPP p q = mulPP q p.
Lemma mulPP_p_p : ∀ p,
  mulPP p p = p.
Lemma mulPP_distr_addPP : ∀ p q r,
  mulPP (addPP p q) r = addPP (mulPP p r) (mulPP q r).
Lemma mulPP_distr_addPPr : ∀ p q r,
  mulPP r (addPP p q) = addPP (mulPP r p) (mulPP r q).
Lemma mulPP_is_poly : ∀ p q,
  is_poly (mulPP p q).
Lemma mulPP_mono_cons : ∀ x m,
  is_mono (x :: m) →
  mulPP [[x]] [m] = [x :: m].
Lemma addPP_poly_cons : ∀ m p,
  is_poly (m :: p) →
  addPP [m] p = m :: p.
Hint Resolve addPP_is_poly mulPP_is_poly.


Lemma mulPP_addPP_1 : ∀ p q r,
  mulPP (addPP (mulPP p q) r) (addPP [[]] q) =
  mulPP (addPP [[]] q) r.
Lemma partition_filter_fst {X} p l :
  fst (partition p l) = @filter X p l.
Lemma partition_filter_fst' : ∀ {X} p (l t f : list X),
    partition p l = (t, f) →
    t = @filter X p l .
Definition neg {X:Type} := fun (f:X→bool) ⇒ fun (a:X) ⇒ (negb (f a)).
Lemma neg_true_false : ∀ {X} (p:X→bool) (a:X),
  (p a) = true ↔ neg p a = false.
Lemma neg_false_true : ∀ {X} (p:X→bool) (a:X),
  (p a) = false ↔ neg p a = true.
Lemma partition_filter_snd {X} p l :
  snd (partition p l) = @filter X (neg p) l.
Lemma partition_filter_snd' : ∀ {X} p (l t f : list X),
  partition p l = (t, f) →
  f = @filter X (neg p) l.
Lemma incl_vars_addPP : ∀ xs p q,
  incl (vars p) xs ∧ incl (vars q) xs ↔ incl (vars (addPP p q)) xs.
Lemma incl_vars_mulPP : ∀ xs p q,
  incl (vars p) xs ∧ incl (vars q) xs ↔ incl (vars (mulPP p q)) xs.
Lemma incl_nil : ∀ {X:Type} (l:list X),
  incl l [] ↔ l = [].
Lemma part_add_eq : ∀ f p l r,
  is_poly p →
  partition f p = (l, r) →
  p = addPP l r.
Lemma part_fst_true : ∀ X p (l t f : list X),
  partition p l = (t, f) →
  (∀ a, In a t → p a = true).
Lemma part_snd_false : ∀ X p (x t f : list X),
  partition p x = (t, f) →
  (∀ a, In a f → p a = false).
Lemma part_Sorted : ∀ {X:Type} (c:X→X→Prop) f p,
  Sorted c p →
  ∀ l r, partition f p = (l, r) →
  Sorted c l ∧ Sorted c r.
Lemma part_is_poly : ∀ f p l r,
  is_poly p →
  partition f p = (l, r) →
  is_poly l ∧ is_poly r.
```

```
Lemma addPP_cons : ∀ (m:mono) (p:poly),
  HdRel (fun m n ⇒ lex compare m n = Lt) m p →
  addPP [m] p = m :: p.
```

Chapter 6

Library B_Unification.poly_unif

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

Require Export poly.

Definition repl := (prod var poly).

Definition subst := list repl.

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

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

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

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

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

Lemma substP_distr_addPP : ∀ p q s,
  substP s (addPP p q) = addPP (substP s p) (substP s q).
Lemma substM_cons : ∀ x m,
  ¬ In x m →
  ∀ q s, substM ((x, q) :: 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.
Lemma substP_1 : ∀ s,
  substP s [[]] = [[]].
Lemma substP_is_poly : ∀ s p,
  is_poly (substP s p).
Hint Resolve substP_is_poly.
Definition unifier (s : subst) (p : poly) : Prop :=
  substP s p = [].
Definition unifiable (p : poly) : Prop :=
  ∃ s, unifier s p.
Definition subst_comp (s t u : subst) : Prop :=
  ∀ x,
  substP t (substP s [[x]]) = substP u [[x]].
Definition more_general (s t : subst) : Prop :=
  ∃ u, subst_comp s u t.
Definition mgu (s : subst) (p : poly) : Prop :=
  unifier s p ∧
  ∀ t,
  unifier t p →
  more_general s t.
Definition reprod_unif (s : subst) (p : poly) : Prop :=
  unifier s p ∧
  ∀ t,
  unifier t p →
  subst_comp s t t.
Lemma subst_comp_poly : ∀ s t u,
  (∀ x, substP t (substP s [[x]]) = substP u [[x]]) →
  ∀ p,
  is_poly p →
  substP t (substP s p) = substP u p.
Lemma reprod_is_mgu : ∀ p s,
  reprod_unif s p →
  mgu s p.
```

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

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

Lemma empty_unifier : unifier [] [].

Lemma empty_mgu : mgu [] [].

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

Chapter 7

Library B_Unification.sve

7.1 Intro

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

7.2 Eliminating Variables

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

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

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

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

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

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

```
Lemma elim_var_not_in_rem : ∀ x p r,
  elim_var x p = r →
  (∀ m, In m r → ¬ In x m).

Lemma elim_var_poly : ∀ x p,
  is_poly (elim_var x p).

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

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).

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

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

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)).

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').

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)).
Lemma p_map_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)).

Lemma elim_var_permutation : ∀ p x,
  is_poly p →
  (∀ m, In m p → In x m) →
  Permutation p (mulPP [[x]] (elim_var x p)).

Lemma elim_var_mul : ∀ x p,
  is_poly p →
  (∀ m, In m p → In x m) →
  p = mulPP [[x]] (elim_var x p).

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

Lemma part_var_eq_in : ∀ x p i o,
  partition (has_var x) p = (i, o) →
  ((∀ m, In m i → In x m) ∧
   (∀ m, In m o → ¬ In x m)).

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 $p = x \times q + r$.

```
Lemma div_eq : ∀ x p q r,
  is_poly p →
  div_by_var x p = (q, r) →
  p = addPP (mulPP [[x]] q) r.

Lemma has_var_in : ∀ x m,
  In x m → has_var x m = true.

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)).
```

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

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

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

Lemma build_poly_is_poly : ∀ q r,
  is_poly (build_poly q r).

Lemma div_build_unif : ∀ x p q r s,
  is_poly p →
  div_by_var x p = (q, r) →
  unifier s p →
  unifier s (build_poly q r).

Lemma div_by_var_nil : ∀ x q r,
  div_by_var x [] = (q, r) →
  q = [] ∧ r = [].

Hint Unfold vars div_by_var elim_var make_poly MonoSort.sort.
Hint Resolve div_by_var_nil.

Lemma incl_not_in : ∀ A a (l m : list A)
  (Aeq_dec : ∀ (a b : A), {a = b}+{a ≠ b}),
  incl l (a :: m) →
  ¬ In a l →
  incl l m.

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

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.
```

7.3 Building Substitutions

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

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

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.

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) →
  inDom x s = false →
  ∀ t, unifier t p →
            subst_comp (build_subst s x q r) t t.

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) →
  inDom x s = false →
  reprod_unif (build_subst s x q r) p.
```

7.4 Recursive Algorithm

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

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

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

7.5 Correctness

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

```
Lemma sve_in_vars_in_unif : ∀ xs y p,
  NoDup xs →
  incl (vars p) xs →
  is_poly p →
  ¬ In y xs →
  ∀ s, sveVars xs p = Some s →
             inDom y s = false.
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.
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.

Lemma sveVars_correct : ∀ (p : poly),
  is_poly p →
  match sveVars (vars p) p with
  | Some s ⇒ mgu s p
  | None ⇒ ¬ unifiable p
  end.

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