A Coq Formalization of Boolean Unification

Dan Dougherty
Worcester Polytechnic Institute,
Worcester, MA, U.S.A.
dd@wpi.edu

April 2019

A verified implementation of two (well-known) algorithms for unification modulo the theory of Boolean rings: Lowenheim's method and the method of Successive Variable Elimination.

Contents

Chapter 1

Library BooleanRings.Introduction

1.1 Introduction

The axioms

$$\begin{array}{cc} x + y \approx y + x, & x * y \approx y * x, \\ (x + y) + z \approx x + (y + z), & (x * y) * z \approx x * (y * z), \\ x + x \approx 0, & x * x \approx x, \\ 0 + x \approx x, & 0 * x \approx 0, \\ x * (y + z) \approx (x * y) + (x * z), & 1 * x \approx x \end{array}$$

The axioms other than the last are precisely the axioms for a ring, with the omission of an additive inverse and the inclusion of the axiom $x+x = 0$. But in the presence of an additive inverse, the idempotence axiom $x^2 = x$ entails $x + x = x$, and so the additive inverse is in fact the identity. Proof: expand $x+x$ as $(x+x)(x+x) = xx+xx+xx+xx = x+x+x+x$. Cancellation yields $0 = x + x$. Presenting the theory as above, with $x + x = 0$ as an axiom, rather than using a separate additive inverse operator, thus simplifies a formal development: there is one fewer function symbol in the signature.

Two easy but significant consequences of the axioms:

(i) commutativity of multiplication: $xy = yx$

Proof: $x+y = (x+y)(x+y) = xx+xy+yx+yy = x+xy+yx+y$. Cancellation yields $0 = xy + yx$ and

(ii) the fact that every element is a 0-divisor

Proof: $x(1 + x) = x + xx = x + x = 0$.

Chapter 2

Library BooleanRings.Utilities

```
Require Export Bool.
Require Export Omega.
Require Export List.
Export ListNotations.
From Coq Require Export ListDec.
Require Export Morphisms.
Require Export Classical.

Require Export Base.

Global Set Implicit Arguments.
Global Unset Strict Implicit.
```

2.1 Decidability stuff

2.1.1 Working with decision

```
Lemma neq_if_eq_dec (U: Type) (x y: nat) (u v: U) :
  x≠y →
  (if decision (x=y) then u else v) = v.
Proof.
  intros Hneq.
  decide (x=y); congruence.
Qed.
Arguments neq_if_eq_dec {U} x y u v.

Lemma eq_if_eq_dec (U: Type) (x y: nat) (u v: U) :
  x=y →
  (if decision (x=y) then u else v) = u.
Proof.
```

```
  intros Hneq.
  decide (x=y); congruence.
Qed.
Arguments eq_if_eq_dec {U} x y u v.

Lemma neq_if_neq_dec (U: Type) (x y: nat) (u v: U) :
  x≠y →
  (if decision (x≠y) then u else v) = u.
Proof.
  intros Hneq.
  decide (x≠y); easy.
Qed.
Arguments neq_if_neq_dec {U} x y u v.

Lemma eq_if_neq_dec (U: Type) (x y: nat) (u v: U) :
  x=y →
  (if decision (x≠y) then u else v) = v.
Proof.
  intros Hneq.
  decide (x≠y); congruence.
Qed.
Arguments eq_if_neq_dec {U} x y u v.

Lemma y_if_y_dec {U: Type} (P: Prop) {D: dec P} (u v: U) :
  P →
  (if decision P then u else v) = u.
Proof.
  intros Hneq.
  decide P; congruence.
Qed.
Arguments y_if_y_dec {U} P {D} u v.

Lemma n_if_y_dec {U: Type} (P: Prop) {D: dec P} (u v: U) :
  ¬ P →
  (if decision P then u else v) = v.
Proof.
  intros Hneq.
  decide P; congruence.
Qed.
Arguments n_if_y_dec {U} P {D} u v.
```

2.1.2 Boolean Reflection

Boolean equality on nat Nat.eqb is also aliased as beq_nat

```
Definition eqb_nat := Nat.eqb.
```

```
Lemma beq_reflect : ∀ (x y : nat), reflect (x = y) (x =? y).
Proof.
  intros x y.
  apply iff_reflect. symmetry. apply beq_nat_true_iff.
Qed.

Lemma eqb_nat_reflect : ∀ (x y : nat), reflect (x = y) (x =? y).
Proof.
  intros x y.
  apply iff_reflect. symmetry. apply beq_nat_true_iff.
Qed.

Lemma blt_reflect : ∀ (x y : nat), reflect (x < y) (x <? y).
Proof.
  intros x y.
  apply iff_reflect. symmetry. apply Nat.ltb_lt.
Qed.

Lemma ble_reflect : ∀ (x y : nat), reflect (x ≤ y) (x <=? y).
Proof.
  intros x y.
  apply iff_reflect. symmetry. apply Nat.leb_le.
Qed.
```

2.1.3 Connecting sumbool with booleans

```
Definition sumbool_to_bool :
  ∀ P Q : Prop, {P} + {Q} → bool :=
  fun P Q sb ⇒ if sb then true else false.

Arguments sumbool_to_bool {P} {Q} _.

Lemma sumbool_to_bool_correct_left P Q sb :
  (@sumbool_to_bool P Q sb) = true → P.
Proof.
 intros H.
    unfold sumbool_to_bool in H.
    destruct sb as [y | n]; easy.
Qed.

Lemma sumbool_to_bool_correct_right P Q sb :
  (@sumbool_to_bool P Q sb) = false → Q.
Proof.
  intros H.
  unfold sumbool_to_bool in H.
  destruct sb as [y | n]; easy.
Qed.
```

```
Definition dec_to_bool :
  ∀ P : Prop, {P} + {¬ P} → bool :=
  fun P sb ⇒ @sumbool_to_bool P (¬P) sb.

Arguments dec_to_bool {P} _.

Lemma dec_to_bool_correct_true P sb :
  (@dec_to_bool P sb) = true ↔ P.
Proof.
  split.
  - unfold dec_to_bool.
    apply sumbool_to_bool_correct_left.
  - intros H.
    unfold dec_to_bool.
    apply not_false_is_true; intros H1.
    assert (H2:= sumbool_to_bool_correct_right H1 ).
    easy.
Qed.

Lemma dec_to_bool_correct_false P sb :
  (@dec_to_bool P sb) = false ↔ (P → False).
Proof.
  assert (H1:= dec_to_bool_correct_true sb).
  assert (H2:= not_true_iff_false (dec_to_bool sb)).
  firstorder.
Qed.

Global Unset Implicit Arguments.

Instance bool_eq_dec :
  eq_dec bool.
Proof.
  intros x y. hnf. decide equality.
Defined.

Instance nat_eq_dec :
  eq_dec nat.
Proof.
  intros x y. hnf. decide equality.
Defined.

Instance nat_le_dec (x y : nat) : dec (x ≤ y) :=
  le_dec x y.
```

2.2 Classical Logic

Classical contrapositive `Lemma classical_contra :`

```
  ∀ p q, (¬p → ¬q) → q →p.
Proof.
  intros.
  apply NNPP. firstorder.
Qed.

Lemma some_inj {X: Type} (x y: X) :
  (Some x = Some y) → x = y.
Proof.
  intros. congruence.
Qed.
Hint Resolve some_inj.

Lemma neq_comm : ∀ T (x y: T),
    x ≠ y → y ≠ x.
Proof.
  unfold not in *.
  intros.
  firstorder.
Qed.
Arguments neq_comm {T} x y.

Lemma eqb_comm : ∀ (x y: nat),
    x =? y = true → y =? x = true.
Proof.
  intros x y H.
  assert (H1:= beq_reflect y x).
  inversion H1.
  - easy.
  - rewrite (Nat.eqb_eq x y ) in H.
    auto.
Qed.

Lemma neqb_comm : ∀ (x y: nat),
    x =? y =false → y =? x =false.
Proof.
  intros x y H.
  destruct (beq_reflect x y) as [si|non].
  - inversion H.
  - unfold not in non.
    rewrite Nat.eqb_neq.
    unfold not.
    auto.
Qed.

Lemma beq_nat_false' : ∀ n m : nat, n ≠ m → (n =? m) = false.
```

```
Proof.
  apply beq_nat_false_iff.
Qed.

Ltac beqtac := rewrite ← beq_nat_refl; reflexivity.
```

2.3 Boolean list membership

```
Definition inb (n: nat) (l: list nat) : bool :=
  existsb (fun x ⇒ Nat.eqb x n) l.

Lemma inb_nil (n: nat) : inb n [] = false.
Proof.
  reflexivity. Qed.
Hint Resolve inb_nil.

Lemma inb_head (n: nat)(xs: list nat) : inb n (n::xs) = true.
Proof.
  unfold inb.
  apply existsb_exists.
  ∃ n.
  split.
  simpl; now left.
  now rewrite ← (beq_nat_refl n).
Qed.
Hint Resolve inb_head.

Lemma inb_tail (n m : nat)(xs: list nat) :
  n ≠ m → inb n (m::xs) = true → inb n xs = true.
Proof.
  unfold inb.
  intros H1 H2.
  apply existsb_exists in H2.
  apply existsb_exists.
  destruct H2 as [x' [H21 H22]].
  apply beq_nat_true in H22.
  rewrite H22 in H21.
  assert (b:= in_cons_neq H21 H1 ).
  ∃ n.
  split. easy.
  symmetry. apply beq_nat_refl.
Qed.
Hint Resolve inb_tail.

Lemma inb_false1:
```

```
  ∀ (n m : nat) (xs : list nat),
    inb n (m :: xs) = false → n ≠ m.
Proof.
  intros n m xs H.
  simpl in H.
  assert (a:= orb_false_iff (m=?n) (inb n xs)).
  destruct a as [a1 a2].
  assert (b:= a1 H).
  destruct b as [b1 b2].
  apply neq_comm.
  now apply beq_nat_false.
Qed.
Hint Resolve inb_false1.

Lemma inb_false2:
  ∀ (n m : nat) (xs : list nat),
    inb n (m :: xs) = false → inb n xs = false.
Proof.
  intros n m xs H.
  assert (a:= orb_false_iff (m=?n) (inb n xs) ).
  destruct a as [a1 a2].
  simpl in H.
  assert (b:= a1 H).
  now destruct b as [b1 b2].
Qed.
Hint Resolve inb_false2.

Lemma natlist_reflect : ∀ (n: nat) (l : list nat),
    reflect (In n l) (inb n l).
Proof.
intros n l.
apply iff_reflect.
split.
- intros H.
  induction l.
  inversion H.
  destruct (beq_reflect n a).
  + rewrite e. auto.
  + inversion H.
    congruence.
    apply existsb_exists.
    ∃ n.
    split.
    now apply in_cons.
```

```
    symmetry; now apply beq_nat_refl.
- intros.
  unfold inb in H; apply existsb_exists in H.
  destruct H as [x [H1 H2]].
  apply beq_nat_true in H2; subst; easy.
Qed.
```

2.4 None and Some

```
Lemma None_not_Some {T U: Type} (f : U → option T) (x: U):
  (f x) = None → (∀ (t: T), ¬ (f x) = Some t).
Proof.
  intros H1 H2 H3.
  congruence.
Qed.

Lemma Some_not_None {T U: Type} (f : U → option T) (x: U) (t: T):
  (f x) = Some t → ¬ (f x = None).
Proof.
  intros H1 H2.
  congruence.
Qed.

Lemma not_None_Some {T U: Type} (f : U → option T) (x: U) :
  ¬ (f x = None) → ∃ t : T, f x = Some t.
Proof.
  intros H.
  destruct (f x) as [t | ].
  - ∃ t; easy.
  - congruence.
Qed.

Lemma not_Some_None {T U: Type} (f : U → option T) (x: U) :
 ( ¬ ∃ t : T, f x = Some t) → f x = None.
Proof.
  apply classical_contra.
  intros H.
  apply not_None_Some in H.
  tauto.
Qed.
```

2.5 Option Map: Reasoning Backwards

```
Lemma option_map_reflect_None {A B : Type} (f: A → B) (oA: option A):
  option_map f oA = None →
  oA = None.
Proof.
  intros H.
  destruct oA .
  inversion H.
  easy.
Qed.
```

```
Lemma option_map_reflect_Some {A B : Type} (f: A → B) (oA: option A) (b: B):
  option_map f oA = Some b →
  ∃ (a: A), (oA = Some a) ∧ (f a) = b.
Proof.
  intros H.
  destruct oA as [a | ].
  - ∃ a.
    simpl in H.
    firstorder.
  - inversion H.
Qed.
```

Classical Logic stuff
A better name

```
Definition Excluded_Middle := classic.
```

use in reasoning by cases

```
Lemma strong_disjunct_L (P Q: Prop):
  (P ∨ Q) →
  (P ∨ (¬P ∧ Q)).
Proof.
  intros H.
  assert (PnotP:= classic P).
  tauto.
Qed.
```

```
Lemma strong_disjunct_R (P Q: Prop):
  (P ∨ Q) →
  ((P ∧ ¬Q) ∨ Q).
Proof.
  intros H.
  assert (PnotP:= classic P).
```

```
  tauto.
Qed.

  Lemma in_sing_refl :
    ∀ {X : Type} (x :X),
      x el [x].
    Proof.
      simpl. now left.
  Qed.
```

2.6 List Utilities

From Guillaume Claret's CUnit module.

useful in testing

```
Fixpoint map_pair {A B C : Type} (f : A → B → C) (l : list (A × B))
  : list C :=
  match l with
  | [] ⇒ []
  | (a, b) :: l ⇒ f a b :: map_pair f l
  end.

Fixpoint cons_all {T: Type} (new: T) (lsls: list (list T)) : (list (list T)):=
  match lsls with
  | [] ⇒ []
  | ls :: rest ⇒ (new :: ls) :: (cons_all new rest)
  end.

Lemma existsb_find {T: Type} (f: T → bool) (l : list T) :
  existsb f l = true →
  ∃ (a: T), find f l = Some a.
Proof.
  intros H.
  apply NNPP.
  intros H1.
  apply not_Some_None in H1.
  assert (A1:= find_none f l).
  assert (A2:= A1 H1).
  assert (A3:= existsb_exists f l).
  destruct A3 as [A31 A32].
  assert (A4:= A31 H).
  destruct A4 as [t A41]. destruct A41 as [A411 A412].
  assert (A21:= A2 t A411).
  rewrite A412 in A21.
```

```
  congruence.
Qed.

Lemma find_existsb {T: Type} (f: T → bool) (l : list T) :
  (∃ (a: T), find f l = Some a) →
  existsb f l = true .
  Proof.
    intros H.
    destruct H as [a H1].
    apply existsb_exists.
    assert (A1:= find_some f l).
    ∃ a.
    firstorder.
Qed.

  Lemma append_nil1 :
    ∀ (A: Type) (l1 l2: list A),
      l1 ++ l2 = [] → l1 = [].
  Proof.
    intros T l1 l2 H.
    destruct l1 as [|k].
    - reflexivity.
    - inversion H.
  Qed.

  Lemma append_nil2 :
    ∀ (A: Type) (l1 l2: list A),
      l1 ++ l2 = [] → l2 = [].
  Proof.
    intros T l1 l2 H.
    - destruct l2 as [|k].
      + reflexivity.
      + assert (a: l1 = []).
        apply append_nil1 in H; easy.
        rewrite a in H.
        inversion H.
  Qed.

  Lemma append_nil:
    ∀ (A: Type) (l1 l2: list A),
      l1 ++ l2 = [] →
      l1 = [] ∧ l2 = [].
  Proof.
    intros T l1 l2 H.
    split.
```

```
  apply append_nil1 with (l2 := l2); easy.
  apply append_nil2 with (l1 := l1); easy.
Qed.

Section MoreEqui.
```

Mirroring some lemmas in List, but with undup involved

```
  Variable X : Type.
  Context {eq_X_dec : eq_dec X}.

Lemma equi_nil (l: list X) :
  equi l nil →
  l = nil.
Proof.
  intros H.
  unfold equi in H.
  destruct H.
  now assert (a:= incl_nil_eq H).
Qed.
Hint Resolve equi_nil.

Lemma undup_nil l :
  undup l = nil → l = nil.
Proof.
  intros H.
  assert (a:= undup_id_equi l).
  rewrite H in a.
  symmetry in a.
  now apply equi_nil.
Qed.
Hint Resolve undup_nil.

Lemma undup_no_new_elts l x :
  x el (undup l) → x el l.
Proof.
  intros H.
  now rewrite (undup_id_equi l) in H.
Qed.
Hint Resolve undup_no_new_elts.

Lemma undup_keep_elts l x :
  x el l → x el (undup l).
Proof.
  intros H.
  now rewrite ← (undup_id_equi l) in H.
Qed.
```

```
  Hint Resolve undup_keep_elts.

  Lemma undup_same_elts l x :
    x el l ↔ x el (undup l).
  Proof.
    split.
    - intros H. exact (undup_keep_elts l x H).
    - intros H. exact (undup_no_new_elts l x H).
  Qed.
  Hint Resolve undup_same_elts.

  Lemma undup_in_app_or l m x :
    x el (undup (l ++ m)) → x el l ∨ x el m.
  Proof.
    intros H.
    assert (a:= (undup_no_new_elts (l++m) x H)).
    exact (in_app_or l m x a).
  Qed.
  Hint Resolve undup_in_app_or.

  Lemma undup_in_or_app :
    ∀ (l m:list X) (x:X), In x l ∨ In x m → In x (undup (l ++ m)).
  Proof.
    intros l m x H.
    assert (H1:= in_or_app l m x H).
    now apply undup_keep_elts.
  Qed.
  Hint Resolve undup_in_or_app.

  Lemma undup_in_app_iff :
    ∀ l l' (x:X), In x (undup (l++l')) ↔ In x l ∨ In x l'.
  Proof.
    intros l l' x.
    split.
    -intros H.
     assert (a:= (undup_no_new_elts (l++l') x H )).
     now apply (in_app_iff).
    - intros H.
      exact (undup_in_or_app l l' x H).
  Qed.
  Hint Resolve undup_in_app_iff.

End MoreEqui.

  Lemma incl_In {T: Type} (x: T) (l: list T) :
    [x] «= l → In x l.
  Proof. auto. Qed.
```

```
  Lemma incl_el {T: Type} (x: T) (l: list T) :
    [x] <<= l <-> x el l.
  Proof.
    unfold incl.
    split.
  - intros H.
    now apply (H x).
  - intros H1 a H2.
    simpl in H2.
    destruct H2.
    now rewrite <- H.
    easy.
Qed.
```

```
    Lemma incl_cons {T: Type} (x: T) (l: list T) :
  l <<= x :: l.
Proof.
  auto.
Qed.
```

```
Lemma incl_neq_head {T: Type} (x: T) (l1 l2 : list T):
  l1 <<= x::l2 -> ~ x el l1 -> l1 <<= l2.
Proof.
  intros H1 H2.
  now apply (incl_rcons H1).
Qed.
```

```
Lemma not_in_rem {T: Type} {_: eq_dec T} (x: T) (l: list T) :
  ~ x el (rem l x).
Proof.
  apply rem_not_in; left; easy.
Qed.
```

```
Lemma el_incl {T: Type} {_: eq_dec T} (x: T) (l1 l2: list T) :
  x el l1 -> l1 <<= l2 -> x el l2.
Proof.
  auto.
Qed.
```

More refined version of not_in_rem

```
Lemma not_in_rem_incl {T: Type} {_: eq_dec T} (x: T) (l1 l2: list T) :
  l1 <<= (rem l2 x) -> ~ x el l1.
Proof.
  intros H0 H1.
  assert (A:= el_incl x l1 (rem l2 x) H1 H0).
  now assert (B:= not_in_rem x l2).
```

```
Qed.
```

rem and cons are dual

```
Lemma cons_rem_swap {T: Type} {_: eq_dec T} (x: T) (l r : list T):
  l «= (x :: r) →
  (rem l x) «= r.
Proof.
  intros H.
  assert (A:= @rem_mono T _ l (x::r) x H).
  assert (B: rem (x::r) x «= r).
  - now apply rem_cons.
  - now apply incl_tran with (m:= rem (x::r) x).
Qed.

Lemma rem_cons_swap {T: Type} {_: eq_dec T} (x: T) (l r : list T):
  (rem l x) «= r →
  l «= (x :: r).
Proof.
  intros H.
  assert (A: x :: (rem l x) «= x :: r).
  { now apply incl_shift. }
  rewrite ← rem_equi in A.
  assert (B:= incl_cons x l).
  now apply incl_tran with (x :: l).
Qed.

Lemma rem_app_hom x l r :
  (rem l x) ++ (rem r x) = rem (l ++ r) x.
   Proof.
     unfold rem; rewrite filter_app; auto.
  Qed.

Lemma el_app_L {T: Type} (x: T) (l r : list T):
  x el l → x el (l ++ r).
  Proof. auto.
Qed.

  Lemma el_app_R {T: Type} (x: T) (l r : list T):
  x el r → x el (l ++ r).
  Proof. auto.
Qed.
```

Chapter 3

Library BooleanRings.TheoryDef

```
Require Export Bool.
Require Export Omega.
Require Export List.
Export ListNotations.
Require Export Setoid.
Require Export Morphisms.
Require Export Relation_Definitions.
Require Export PeanoNat.
Require Export Ring.
Require Export Coq.btauto.Algebra.
Require Export DecidabilityAndReflection.
Require Export Utilities.
```

- Author: Dan Dougherty, WPI
- Based a treatment in "Term Rewriting and All That", by Franz Baader and Tobias Nipkow

3.1 Introduction

A Boolean ring is a ring satifying the identity $x * x = x$. From this follows the identity $x + x = 0$. It is convenient to add this identity as an axiom.

3.2 Variables and Bterms

```
Notation var := nat (only parsing).
Inductive bterm: Type :=
| T0 : bterm
```

```
| T1 : bterm
| V : var → bterm
| A : bterm → bterm → bterm
| M : bterm → bterm → bterm.
```

Notation

Multiplication binds more tightly than addition

```
Notation "x +' y" := (A x y) (at level 50, left associativity).
Notation "x *' y" := (M x y) (at level 49, left associativity).
```

3.3 Axioms

```
Reserved Notation "x == y" (at level 70).
Inductive eqv : bterm → bterm → Prop :=
| assocA: ∀ x y z, (x +' y) +' z == x +' (y +' z)
| invA: ∀ x, x +' x == T0
| idA: ∀ x, T0 +' x == x
| comA: ∀ x y, x +' y == y +' x

| assocM: ∀ x y z, (x *' y) *' z == x *' (y *' z)
| ord2M: ∀ x, x *' x == x
| idM: ∀ x, T1 *' x == x

| dist: ∀ x y z, (y +' z) *' x == (y *' x) +' (z *' x)
| dist': ∀ x y z, x *' (y +' z) == ( x *' y) +' (x *' z)

| eqv_ref: ∀ x, x == x
| eqv_sym: ∀ x y, x == y → y == x
| eqv_trans: ∀ x y z, x == y → y == z → x == z

| A_compat : ∀ x x' , x == x' → ∀ y y' ,
       y == y' → x +' y == x' +' y'
| M_compat : ∀ x x' , x == x' → ∀ y y' ,
       y == y' → x *' y == x' *' y'
where "x == y" := (eqv x y).

Hint Constructors eqv.
```

3.4 Make eqv suitable for setoid rewriting

Eqv is an equivalence relation

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

Eqv is a congruence wrt addition and multiplication

```
Add Parametric Morphism : A with
      signature eqv ==> eqv ==> eqv as A_mor.
  exact A_compat.
Qed.
Add Parametric Morphism : M with
      signature eqv ==> eqv ==> eqv as M_mor.
  exact M_compat.
Qed.
```

3.5 Decidability of equality on bterms

```
Definition var_eq_dec := nat_eq_dec.
Instance bterm_eq_dec :
  eq_dec bterm.
Proof.
  unfold dec.
  decide equality.
  apply var_eq_dec.
Qed.
Hint Resolve bterm_eq_dec.
```

Boolean bterm equality

```
Definition bterm_eqb t1 t2 :=
  if (decision (t1 = t2) )
  then true else false.
Lemma btermeq_if t1 t2 :
  t1 = t2 → bterm_eqb t1 t2 = true.
Proof.
```

```
  intros H.
  unfold bterm_eqb.
  now rewrite (y_if_y_dec (t1 = t2) true false).
Qed.

Lemma btermeq_onlyif t1 t2 :
  bterm_eqb t1 t2 = true → t1 = t2.
Proof.
  intros H.
  unfold bterm_eqb in H.
  decide (t1 = t2);
    easy.
Qed.

Lemma btermeq_reflect : ∀ (t1 t2 : bterm),
    reflect (t1 = t2) (bterm_eqb t1 t2).
Proof.
  intros t1 t2.
  apply iff_reflect.
  split.
  apply btermeq_if.
  apply btermeq_onlyif.
Qed.
```

Chapter 4

Library BooleanRings.BTermsUtilities

```
Require Export Omega.
Require Export List.
Export ListNotations.
Require Export Setoid.
Require Export Morphisms.
Require Export Relation_Definitions.
Require Export Ring.
Require Export Utilities.
Require Export TheoryDef.
```

4.1 Useful Little Results

4.1.1 Identities, distributivity, and absorption apply on either side

```
Lemma idA' (x: bterm) :
  x +' T0 == x .
Proof. rewrite comA; auto.
Qed.
```

4.1.2 Additive cancellation on the left and the right

```
Lemma cancelA_R :
  ∀ x y z, x +' z == y +' z → x == y.
Proof.
  intros x y z H.
  cut ((x +' z) +' z == (y +' z) +' z).
  - intros H0. repeat rewrite assocA in H0.
    repeat rewrite invA in H0.
    repeat rewrite idA' in H0; easy.
```

```
  - rewrite H; easy.
Qed.

Lemma cancelA_L :
  ∀ x y z, z +' x == z +' y → x == y.
Proof.
  intros x y z H.
  rewrite comA in H at 1.
  rewrite (comA z y) in H.
  apply cancelA_R with (z := z); easy.
Qed.
```

4.1.3 Equivalence and 0-testing

```
Lemma eqv_eqv0 (s t: bterm) :
  s == t ↔ (s +' t) == T0.
Proof.
  split.
  - intros H.
    rewrite H. apply invA.
  - intros H.
    assert (a : s +' t +' t == T0 +' t).
    + now rewrite H.
    + rewrite assocA in a.
      rewrite idA in a.
      rewrite invA in a.
      now rewrite idA' in a.
Qed.
```

4.1.4 Boolean rings are commutative rings

```
Lemma comM: ∀ x y, (x *' y ) == (y *' x).
Proof.
  intros x y.
  assert (a:= ord2M (x +'y) ).
  assert (b: ( (x +' y) *' (x +' y) ==
                 x +' (x *' y) +' (y *' x) +' y)).
  - rewrite dist.
    rewrite dist'. rewrite dist'.
    rewrite ord2M. rewrite ord2M.
    repeat rewrite assocA. easy.
  - rewrite a in b.
    apply cancelA_R in b.
```

```
    rewrite assocA in b.
    assert (c: x +'x == x +' x +' (x *' y +' y *' x)).
    + rewrite assocA.
      now rewrite ← b.
    + rewrite invA in c.
      rewrite idA in c.
      apply eqv_eqv0.
      now symmetry.
Qed.
Lemma idM' (x: bterm) :
  x *' T1 == x .
Proof. rewrite comM. apply idM.
Qed.
```

0 absorbs multiplication

```
Lemma zeroM : ∀ x, T0 *' x == T0.
Proof.
  intros x.
  rewrite ← (invA x).
  rewrite dist.
  now rewrite ord2M.
Qed.
Lemma zeroM' (x: bterm) :
  x *' T0 == T0.
Proof. rewrite comM. apply zeroM.
Qed.
```

4.1.5 Everything is a 0-divisor

```
Lemma T0_div :
  ∀ x, x *' (x +' T1) == T0.
Proof.
  intros. rewrite dist'.
  rewrite idM'. rewrite ord2M. apply invA.
Qed.
Lemma T0_div' :
  ∀ x, (x +' T1) *' x == T0.
Proof.
  intros x. rewrite comM.
  apply T0_div.
Qed.
```

4.1.6 Additive inverses are unique

```
Lemma inv_uniqueA' :
  ∀ x y, x +' y == T0 → x == y.
Proof.
  intros.
  assert (x +' y +' y == T0 +' y).
  - now rewrite H.
  - rewrite idA in H0.
    rewrite assocA in H0.
    rewrite invA in H0.
    now rewrite idA' in H0.
Qed.

Lemma inv_uniqueA :
  ∀ x y, x +' y == T0 ↔ x == y.
Proof.
  split.
  - intros H.
    assert (x +' y +' y == T0 +' y).
    + now rewrite H.
    + rewrite idA in H0.
      rewrite assocA in H0.
      rewrite invA in H0.
      now rewrite idA' in H0.
  - intros H.
    rewrite H; easy.
Qed.

Lemma T0_T1 (t: bterm) :
  t == T1 ↔ t +' T1 == T0.
Proof.
  split.
  - intros. now rewrite H.
  - intros. assert (H1: (t +' T1 +' T1 == T1)).
    + now rewrite H.
    + rewrite assocA in H1.
      rewrite invA in H1.
      now rewrite idA' in H1.
Qed.

Lemma T1_T0 (t: bterm) :
  t == T0 ↔ t +' T1 == T1.
Proof.
  split.
```

```
  - intros. now rewrite H.
  - intros. assert (H1: (t +' T1 +' T1 == T0)).
    + now rewrite H.
    + rewrite assocA in H1.
      rewrite invA in H1.
      now rewrite idA' in H1.
Qed.

Hint Resolve idA' comM zeroM' idM' T0_div inv_uniqueA inv_uniqueA
     cancelA_L cancelA_R.
```

4.2 Bterms is a ring

Ring theory expects an explicit operator for "minus". But in the presence of x+x=0, minus is the identity

```
Definition ring_minus: bterm → bterm := fun t ⇒ t.

Lemma ring_minus_compat : ∀ x x' ,
    eqv x x' → eqv (ring_minus x) (ring_minus x').
Proof.
  intros x x' Heqv; now unfold ring_minus.
Qed.

Add Parametric Morphism : ring_minus with
      signature eqv ==> eqv as ring_minus_mor.
  exact ring_minus_compat.
Qed.

Definition Bring : ring_theory T0 T1 A M A ring_minus eqv .
  apply Ring_theory.mk_rt; auto.
Qed.

Add Ring boolean_ring : Bring.
```

4.2.1 Adding x*x=x and x+x=0 to a simplification tactic

This is only partially successful: helpful, but does *not* rewrite terms into their polynomial normal form.

```
Ltac bsimp := repeat (try ring_simplify; try rewrite ord2M; try rewrite invA; auto).

Hint Rewrite ord2M invA : boolring_rules.

Ltac bsimp' := autorewrite with boolring_rules using ring_simplify; auto.
```

4.3 Variables occurring

```
Fixpoint is_var (t: bterm) : bool :=
  match t with
  | (V _ ) ⇒ true
  | _ ⇒ false
  end.
```

Use this for now (allowing duplications) since it makes some things easier to prove. Is there a reason to eliminate duplications??

```
Fixpoint vars_bterm (t: bterm) : list var :=
  match t with
  | T0 ⇒ []
  | T1 ⇒ []
  | V x ⇒ [x]
  | A t1 t2 ⇒ ((vars_bterm t1) ++ ( vars_bterm t2))
  | M t1 t2 ⇒ ((vars_bterm t1) ++ ( vars_bterm t2))
  end.
Hint Unfold vars_bterm.

Lemma vars_bterm_A_L: ∀ t1 t2 x,
     In x (vars_bterm t1) →
     In x (vars_bterm (t1 +' t2)).
Proof.
  intros t1 t2 x Hin. simpl.
  apply in_or_app.
  now left.
Qed.

Lemma vars_bterm_incl_A_L t1 t2 :
  vars_bterm t1 «= vars_bterm (t1 +' t2).
Proof.
  unfold incl.
  intros v H.
  now apply vars_bterm_A_L.
Qed.

  Lemma not_vars_bterm_A_L: ∀ t1 t2 x,
    ¬ In x (vars_bterm (t1 +' t2)) →
    ¬ In x (vars_bterm t1) .
Proof.
  intros t1 t2 x Hin.
  assert (a:= vars_bterm_A_L t1 t2 x).
  firstorder.
Qed.
```

```
Lemma vars_bterm_A_R: ∀ t1 t2 x,
    In x (vars_bterm t2) →
    In x (vars_bterm (t1 +' t2)).
Proof.
  intros t1 t2 x Hin. simpl.
  apply in_or_app.
  now right.
Qed.

Lemma vars_bterm_incl_A_R t1 t2 :
  vars_bterm t2 «= vars_bterm (t1 +' t2).
Proof.
  unfold incl.
  intros v H.
  now apply vars_bterm_A_R.
Qed.

Lemma not_vars_bterm_A_R: ∀ t1 t2 x,
    ¬ In x (vars_bterm (t1 +' t2)) →
    ¬ In x (vars_bterm t2) .
Proof.
  intros t1 t2 x Hin.
  assert (a:= vars_bterm_A_R t1 t2 x).
  firstorder.
Qed.

Lemma vars_bterm_M_L: ∀ t1 t2 x,
    In x (vars_bterm t1) →
    In x (vars_bterm (t1 *' t2)).
Proof.
  intros t1 t2 x Hin. simpl.
  apply in_or_app.
  now left.
Qed.

Lemma vars_bterm_incl_M_L t1 t2 :
  vars_bterm t1 «= vars_bterm (t1 *' t2).
Proof.
  unfold incl.
  intros v H.
  now apply vars_bterm_M_L.
Qed.

Lemma not_vars_bterm_M_L: ∀ t1 t2 x,
    ¬ In x (vars_bterm (t1 *' t2)) →
    ¬ In x (vars_bterm t1) .
```

```
Proof.
  intros t1 t2 x Hin.
  assert (a:= vars_bterm_M_L t1 t2 x).
  firstorder.
Qed.

Lemma vars_bterm_M_R: ∀ t1 t2 x,
     In x (vars_bterm t2) →
     In x (vars_bterm (t1 *' t2)).
Proof.
  intros t1 t2 x Hin. simpl.
  apply in_or_app.
  now right.
Qed.

Lemma vars_bterm_incl_M_R t1 t2 :
  vars_bterm t2 «= vars_bterm (t1 *' t2).
Proof.
  unfold incl.
  intros v H.
  now apply vars_bterm_M_R.
Qed.

Lemma not_vars_bterm_M_R: ∀ t1 t2 x,
     ¬ In x (vars_bterm (t1 *' t2)) →
     ¬ In x (vars_bterm t2) .
Proof.
  intros t1 t2 x Hin.
  assert (a:= vars_bterm_M_R t1 t2 x).
  firstorder.
Qed.

Lemma vars_bterm_var (v: var) :
  vars_bterm (V v) = [v].
Proof. easy.
Qed.

Hint Resolve
      vars_bterm_A_L
      vars_bterm_incl_A_L
      not_vars_bterm_A_L
vars_bterm_A_R
vars_bterm_incl_A_R
not_vars_bterm_A_R
vars_bterm_M_L
vars_bterm_incl_M_L
```

not_vars_bterm_M_L
vars_bterm_M_R
vars_bterm_incl_M_R
not_vars_bterm_M_R
vars_bterm_var.

4.4 Ground bterms

Boolean version

```
Fixpoint ground_bterm (t: bterm) : bool :=
  match t with
  | T0 ⇒ true
  | T1 ⇒ true
  | (V _ ) ⇒ false
  | A t1 t2 ⇒ (ground_bterm t1) && (ground_bterm t2)
  | M t1 t2 ⇒ (ground_bterm t1) && (ground_bterm t2)
  end.
```

Prop version

```
Inductive Ground_bterm : bterm → Prop :=
| G0 : Ground_bterm T0
| G1 : Ground_bterm T1
| GA : ∀ t1 t2, Ground_bterm t1 →
                Ground_bterm t2 →
                Ground_bterm (A t1 t2)
| GM : ∀ t1 t2, Ground_bterm t1 →
                Ground_bterm t2 →
                Ground_bterm (M t1 t2).
Hint Constructors Ground_bterm.

Lemma ground_is_Ground t :
  ground_bterm t = true → Ground_bterm t.
Proof.
  induction t as [ | | v | t1 t2| t1 t2]; simpl; auto; easy.
Qed.

Lemma Ground_is_ground t :
  Ground_bterm t → ground_bterm t = true.
Proof.
  intros H.
  induction H; simpl; auto.
Qed.

Lemma Ground_no_vars (t: bterm) :
```

```
  Ground_bterm t → vars_bterm t = [].
Proof.
  intros H.
  induction t as [ | | v | t1 t2| t1 t2].
  - reflexivity.
  - reflexivity.
  - inversion H.
  - inversion H.
    firstorder.
    simpl.
    rewrite H4; rewrite H5.
    reflexivity.
  - inversion H.
    firstorder.
    simpl.
    rewrite H4; rewrite H5.
    reflexivity.
Qed.

Lemma no_vars_Ground (t: bterm) :
  vars_bterm t = [] → Ground_bterm t.
Proof.
  intros H.
  induction t as [ | | v | t1 t2| t1 t2]; simpl; auto.
  - inversion H.
  - simpl in H.
    apply app_eq_nil in H.
    destruct H.
    firstorder.
  - simpl in H.
    apply app_eq_nil in H.
    destruct H.
    firstorder.
Qed.

Lemma Ground_bterm_A_intro t1 t2 :
  Ground_bterm t1 →
  Ground_bterm t2 →
  Ground_bterm (t1 +' t2) .
Proof.
  apply GA.
Qed.
Hint Resolve Ground_bterm_A_intro.

Lemma Ground_bterm_A_elim1 t1 t2 :
```

```
  Ground_bterm (t1 +' t2) →
  Ground_bterm t1.
Proof.
  intros H.
  now inv H.
Qed.
Hint Resolve Ground_bterm_A_elim1.

Lemma Ground_bterm_A_elim2 t1 t2 :
  Ground_bterm (t1 +' t2) →
  Ground_bterm t2.
Proof.
  intros H.
  now inv H.
Qed.
Hint Resolve Ground_bterm_A_elim2.

Lemma Ground_bterm_M_intro t1 t2 :
  Ground_bterm t1 →
  Ground_bterm t2 →
  Ground_bterm (t1 *' t2) .
Proof.
  apply GM.
Qed.
Hint Resolve Ground_bterm_M_intro.

Lemma Ground_bterm_M_elim1 t1 t2 :
  Ground_bterm (t1 *' t2) →
  Ground_bterm t1.
Proof.
  intros H.
  now inv H.
Qed.
Hint Resolve Ground_bterm_M_elim1.

Lemma Ground_bterm_M_elim2 t1 t2 :
  Ground_bterm (t1 *' t2) →
  Ground_bterm t2.
Proof.
  intros H.
  now inv H.
Qed.
Hint Resolve Ground_bterm_M_elim2.

Lemma ground_bterm_A_intro t1 t2 :
  ground_bterm t1 = true →
```

```
  ground_bterm t2 = true →
  ground_bterm (t1 +' t2) = true.
Proof.
  intros H0 H1.
  apply ground_is_Ground in H0.
  apply ground_is_Ground in H1.
  apply Ground_is_ground.
  auto.
Qed.
Hint Resolve ground_bterm_A_intro.

Lemma ground_bterm_A_elim t1 t2 :
  ground_bterm (t1 +' t2) = true →
  ground_bterm t1 = true ∧
  ground_bterm t2 = true.
Proof.
  intros H.
  apply ground_is_Ground in H.
  repeat rewrite Ground_is_ground; inversion H; easy.
Qed.
Hint Resolve ground_bterm_A_elim.

Lemma ground_bterm_M_intro t1 t2 :
  ground_bterm t1 = true →
  ground_bterm t2 = true →
  ground_bterm (t1 *' t2) = true.
Proof.
  intros H0 H1.
  apply ground_is_Ground in H0.
  apply ground_is_Ground in H1.
  apply Ground_is_ground.
  auto.
Qed.
Hint Resolve ground_bterm_M_intro.

Lemma ground_bterm_M_elim t1 t2 :
  ground_bterm (t1 *' t2) = true →
  ground_bterm t1 = true ∧
  ground_bterm t2 = true.
Proof.
  intros H.
  apply ground_is_Ground in H.
  repeat rewrite Ground_is_ground; inversion H; easy.
Qed.
Hint Resolve ground_bterm_M_elim.
```

Chapter 5

Library BooleanRings.Substitutions

```
From Coq Require Export Bool.
Require Export Omega.
Require Export List.
Export ListNotations.
From Coq Require Export Lists.ListSet.
Require Export Basics.
Require Export Setoid.
Require Export Morphisms.
Require Export Relation_Definitions.
Require Export Logic.FunctionalExtensionality.
Require Export TheoryDef.
Require Export BTermsUtilities.
```

5.1 Substitutions

```
Definition sub := var → bterm.

Definition id_sub : sub :=
  fun n ⇒ (V n).

Fixpoint apply_sub (sig: sub) (t : bterm) : bterm :=
  match t with
  | T0 ⇒ t
  | T1 ⇒ t
  | V v ⇒ sig v
  | A t1 t2 ⇒ (apply_sub sig t1) +' (apply_sub sig t2)
  | M t1 t2 ⇒ (apply_sub sig t1) *' (apply_sub sig t2)
  end.

Notation "s ' t" := (apply_sub s t) (at level 50).

Lemma apply_id_sub (t : bterm) :
```

```
  id_sub ' t = t.
Proof.
  induction t as [ | | y | t1 IH1 t2 IH2 | t1 IH1 t2 IH2 ]; simpl; auto;
    rewrite IH1; rewrite IH2; easy.
Qed.

Lemma sub_T0_hom sigma :
  sigma ' T0 == T0.
Proof. auto.
Qed.

Lemma sub_T1_hom sigma :
  sigma ' T1 == T1.
Proof. auto.
Qed.

Lemma sub_A_hom sigma s1 s2 :
  sigma ' (s1 +' s2) == (sigma ' s1) +' (sigma ' s2).
Proof.
  auto.
Qed.
Hint Resolve sub_A_hom.

Lemma sub_M_hom sigma s1 s2 :
  sigma ' (s1 *' s2) == (sigma ' s1) *' (sigma ' s2).
Proof.
  auto.
Qed.
Hint Resolve sub_M_hom.
```

5.1.1 Applying a sub preserves eqv

```
Lemma sub_compat : ∀ (sigma: sub) (t t' : bterm),
    t == t' → (sigma ' t) == (sigma ' t').
Proof.
  intros sigma t t' H.
  induction H; subst; simpl; auto.
  eapply eqv_trans with (sigma ' y); auto.
Qed.

Add Parametric Morphism : apply_sub with
      signature eq ==> eqv ==> eqv as apply_sub_mor.
  exact sub_compat.
Qed.
```

5.1.2 Updating

```
Definition update_sub (sig: sub) (x: var) (u: bterm) (n : var) : bterm :=
  if eqb_nat x n then u else (sig n).
Hint Unfold update_sub.
```

Updates do nothing new off of the updated variable

```
Lemma apply_updated_sub_off_var (sigma: sub) (x y: var) (u: bterm) :
  x ≠ y →
  (update_sub sigma x u) y = sigma y.
Proof.
  intros Hneq.
  simpl. unfold update_sub.
  now rewrite beq_nat_false'.
Qed.

Lemma apply_updated_sub_off (sigma: sub) (x: var) (u t: bterm) :
  ¬ x el (vars_bterm t) →
  (update_sub sigma x u) ' t = sigma ' t .
Proof.
  intros Hnotel.
  induction t as [ | | y | t1 IH1 t2 IH2 | t1 IH1 t2 IH2 ]; simpl; auto.
  - simpl in Hnotel.
    rewrite DM_or in Hnotel.
    destruct Hnotel.
    apply (apply_updated_sub_off_var).
    firstorder.
  - assert (a:= not_vars_bterm_A_L t1 t2 x Hnotel).
    assert (b:= not_vars_bterm_A_R t1 t2 x Hnotel).
    rewrite IH1. rewrite IH2. auto. easy. easy.
  - assert (a:= not_vars_bterm_M_L t1 t2 x Hnotel).
    assert (b:= not_vars_bterm_M_R t1 t2 x Hnotel).
    rewrite IH1. rewrite IH2. auto. easy. easy.
Qed.
Hint Resolve apply_updated_sub_off.
```

Updated sub on the new variable

```
Lemma apply_updated_sub_on (sigma: sub) (x: var) (u: bterm) :
  (update_sub sigma x u) x = u.
Proof.
  intros.
```

```
  simpl. unfold update_sub.
  now rewrite ← beq_nat_refl.
Qed.
Hint Resolve apply_updated_sub_on.
```

5.1.3 Composition

```
Definition compose_sub (second_sub first_sub: sub) : sub :=
  fun x ⇒ second_sub ' (first_sub x).
```

5.1.4 Ground Substitutions

Every bterm in the range of sigma is ground?

```
Definition Ground_sub (sigma : sub) : Prop :=
  ∀ x, Ground_bterm (sigma x).
```

Applying sigma results in a ground bterm?

```
Definition grounding (sigma : sub) (t : bterm) :=
  ground_bterm (sigma ' t).
```

5.1.5 Ordering on Substitutions

Note use of eqv rather than eq

```
Definition sub_leq (lower upper : sub) : Prop :=
  ∃ (tau: sub), ∀ (x: var),
    compose_sub tau lower x == upper x.
Definition sub_leq_strong (lower upper : sub) : Prop :=
  ∀ (x: var),
    compose_sub upper lower x == upper x.
```

5.1.6 Unifiers, MGUs, and All That

```
Definition solves (sigma: sub) (t: bterm) :=
  sigma ' t == T0.
Definition solvable (t: bterm) : Prop :=
  ∃ (sigma: sub), solves sigma t.
```

The most general notion

```
Definition mgu (sigma: sub) (t: bterm): Prop :=
  (solves sigma t) ∧
  ∀ (gamma: sub),
    (solves gamma t) → sub_leq sigma gamma.
```

A strengthening: the mediating sub is the given less-general one

```
Definition mgu_strong (sigma: sub) (t: bterm): Prop :=
  (solves sigma t) ∧
  ∀ (gamma: sub),
    (solves gamma t) → sub_leq_strong sigma gamma.
```

Baader and Nipkow use "reproductive" for our strong mgu property

```
Definition reproductive_solution := mgu_strong.
```

5.2 Facts about subs

```
Lemma compose_bterms' (lower upper mediant : sub) :
  (∀ (x: var), compose_sub mediant lower x == upper x) →
  (∀ (t : bterm), compose_sub mediant lower ' t == upper ' t).
Proof.
  intros H t.
  induction t as [ | | y | t1 IH1 t2 IH2 | t1 IH1 t2 IH2 ]; simpl; auto.
Qed.
```

This one seems easier to use

```
Lemma compose_bterms (lower upper mediant : sub) :
  (∀ (x: var), compose_sub mediant lower x == upper x) →
  (∀ (t : bterm), (mediant ' (lower ' t) == upper ' t)).
Proof.
  intros H t.
  induction t as [ | | y | t1 IH1 t2 IH2 | t1 IH1 t2 IH2 ]; simpl; auto.
  unfold compose_sub in H. now apply (H y).
Qed.
```

```
Lemma sub_Ground (sigma: sub) (t: bterm) :
  Ground_bterm t → ( sigma ' t) = t.
Proof.
  intros H.
  induction H; simpl; auto.
  - now rewrite IHGround_bterm1; rewrite IHGround_bterm2.
  - now rewrite IHGround_bterm1; rewrite IHGround_bterm2.
Qed.
```

```
Lemma sub_ground_bterm (sigma: sub) (t: bterm) :
  (ground_bterm t) = true → ( sigma ' t) = t.
Proof.
  intros H.
  assert (a:= ground_is_Ground t H).
  apply (sub_Ground sigma t a).
Qed.
```

5.2.1 Solvability respects eqv

A consequence of compatibility of apply_sub

```
Lemma solves_compat : ∀ (sigma: sub) (t t': bterm),
    t == t' →
    solves sigma t →
    solves sigma t'.
Proof.
  unfold solves.
  intros sigma t t' Heqv Hsolves.
  now rewrite Heqv in Hsolves.
Qed.
```

5.2.2 Truncating subs

Agrees with sig on vars, identity elsewhere

```
Definition truncate (vars : list var) (sig: sub) : sub :=
    fun x ⇒ if inb x vars then (sig x) else (V x).

Lemma truncate_suffices_help (vars : list var) (t: bterm) (sig: sub) :
  (vars_bterm t) «= vars →
  (truncate vars sig) ' t =
  sig ' t.
  Proof.
    induction t as [ | | y | t1 IH1 t2 IH2 | t1 IH1 t2 IH2 ]; simpl; auto.
    - intros H.
      apply incl_ln in H.
      unfold truncate.
      now destruct (natlist_reflect y vars).
  - intros H.
    assert (a1:= vars_bterm_incl_A_L t1 t2).
    assert (a2:= vars_bterm_incl_A_R t1 t2).
    assert (b1:= incl_tran a1 H).
    assert (b2:= incl_tran a2 H).
    rewrite (IH1 b1).
```

```
    now rewrite (IH2 b2).
  - intros H.
    assert (a1:= vars_bterm_incl_M_L t1 t2).
    assert (a2:= vars_bterm_incl_M_R t1 t2).
    assert (b1:= incl_tran a1 H).
    assert (b2:= incl_tran a2 H).
    rewrite (IH1 b1).
    now rewrite (IH2 b2).
Qed.

Lemma truncate_suffices (t: bterm) (sig: sub) :
  (truncate (vars_bterm t) sig) ' t =
   sig ' t.
Proof.
  now apply truncate_suffices_help.
Qed.

Lemma conclude_truncate (sig1 sig2 : sub) (vars : list var) :
  (∀ (x : var), (x el vars) →
                        sig1 x = sig2 x) →
  (truncate vars sig1) = (truncate vars sig2).
Proof.
  intros H.
  unfold truncate.
  apply functional_extensionality.
  intros x.
  destruct (natlist_reflect x vars).
  - now apply (H x).
  - easy.
Qed.

Lemma localize_sub (sig1 sig2 : sub) (t : bterm) :
  (∀ (x : var), (x el vars_bterm t) →
                        sig1 x = sig2 x) →
  sig1 ' t = sig2 't.
Proof.
  intros.
  assert (a:= conclude_truncate sig1 sig2 (vars_bterm t) H).
  rewrite ← (truncate_suffices t sig1).
  rewrite ← (truncate_suffices t sig2).
  now rewrite a.
Qed.

Lemma mgu_T0 t:
  t == T0 → mgu_strong id_sub t.
```

```
Proof.
  intros H.
  unfold mgu_strong.
  split.
  - unfold solves. simpl. now rewrite apply_id_sub.
  - intros gamma H1.
    now simpl.
Qed.
```

Chapter 6

Library BooleanRings.Substitutions01

From Coq `Require Export` Bool.
`Require Export` Omega.
`Require Export` List.
`Export` *ListNotations*.
From Coq `Require Export` Lists.ListSet.
`Require Export` Basics.
`Require Export` Setoid.
`Require Export` Morphisms.
`Require Export` Relation_Definitions.
`Require Export` Logic.FunctionalExtensionality.

`Require Export` Utilities.
`Require Export` TheoryDef.
`Require Export` BTermsUtilities.
`Require Export` EvalGround.
`Require Export` Substitutions.

6.1 Definitions about 01-subs

`Definition` const0sub (x: var) := T0.
`Hint Unfold` const0sub.

`Inductive` **Is_01sub_list** : **list** var → sub → `Prop` :=
| Is_01sub_listNil : **Is_01sub_list** [] const0sub
| Is_01sub_list0 : ∀ l $sigma$ x,
    Is_01sub_list l $sigma$ →
    Is_01sub_list (x :: l) (update_sub $sigma$ x T0)
| Is_01sub_list1 : ∀ l $sigma$ x,
    Is_01sub_list l $sigma$ →
    Is_01sub_list (x :: l) (update_sub $sigma$ x T1) .

```
Hint Constructors ls_01sub_list.

Definition ls_01sub (sigma : sub) : Prop :=
  ∃ l, ls_01sub_list l sigma.
Hint Unfold ls_01sub.

Fixpoint all_01subs_list (vars : list var) : list sub :=
  match vars with
  | [] ⇒ [const0sub]
  | x :: xs ⇒ let subs := all_01subs_list xs in
                (map (fun s ⇒ update_sub s x T0) subs)
                  ++
                  (map (fun s ⇒ update_sub s x T1) subs)
  end.
Hint Unfold all_01subs_list.

Definition all_01subs_bterm (t : bterm) : list sub :=
  all_01subs_list (vars_bterm t).
```

6.2 Facts about 01 subs

```
Lemma ls_01sub_any_list:
  ∀ (l : list var) (gamma : sub),
    ls_01sub_list l gamma →
    ls_01sub gamma.
Proof.
  intros l gamma H.
  unfold ls_01sub. now ∃ l.
Qed.

Lemma const0_update x :
  const0sub = update_sub const0sub x T0.
Proof.
  apply functional_extensionality.
  intros y.
  unfold const0sub.
  unfold update_sub.
  destruct (beq_reflect x y).
  rewrite e.
  now rewrite ← beq_nat_refl.
  now rewrite beq_nat_false'.
Qed.

Lemma const0_is_01sub_any:
  ∀ (l : list var),
```

```
    ls_01sub_list l const0sub.
Proof.
  intros l.
  unfold ls_01sub.
  induction l as [| x xs IH].
  - apply ls_01sub_listNil.
  - rewrite (const0_update x).
    now apply ls_01sub_list0.
Qed.

Lemma const0_is_01sub :
  ls_01sub const0sub.
Proof.
  unfold ls_01sub.
  ∃ nil.
  now apply (const0_is_01sub_any nil).
Qed.
```

6.2.1 Truncating 01 subs to the vars of a bterm

```
Definition truncate0 (vars : list var) (sig: sub) : sub :=
  fun x ⇒ if inb x vars then (sig x) else T0.

Definition truncate' (t: bterm) (vars : list var) (sig: sub) x :=
  if inb x vars then (sig x)
  else t.

Lemma truncate0_suffices_help (vars : list var) (t: bterm) (sig: sub) :
  (vars_bterm t) «= vars →
  (truncate0 vars sig) ' t =
  sig ' t.
Proof.
  induction t as [ | | y | t1 IH1 t2 IH2 | t1 IH1 t2 IH2 ]; simpl; auto.
  - intros H.
    apply incl_ln in H.
    unfold truncate0.
    now destruct (natlist_reflect y vars).
  - intros H.
    assert (a1:= vars_bterm_incl_A_L t1 t2).
    assert (a2:= vars_bterm_incl_A_R t1 t2).
    assert (b1:= incl_tran a1 H).
    assert (b2:= incl_tran a2 H).
    rewrite (IH1 b1).
    now rewrite (IH2 b2).
```

```
  - intros H.
    assert (a1:= vars_bterm_incl_M_L t1 t2).
    assert (a2:= vars_bterm_incl_M_R t1 t2).
    assert (b1:= incl_tran a1 H).
    assert (b2:= incl_tran a2 H).
    rewrite (IH1 b1).
    now rewrite (IH2 b2).
Qed.

Lemma truncate0_suffices (t: bterm) (sig: sub) :
  (truncate0 (vars_bterm t) sig) ' t =
  sig ' t.
Proof.
  now apply truncate0_suffices_help.
Qed.
```

something's wrong in this proof...too tedious

```
Lemma truncate_update (x : nat) xs gamma :
  truncate0 (x :: xs) gamma =
  update_sub (truncate0 xs gamma) x (gamma x).
Proof.
  apply functional_extensionality. intros y.
  unfold truncate0. unfold update_sub.
  destruct (inb y (x::xs)) eqn : e.
  destruct (beq_reflect x y).
  - rewrite e0.
    now rewrite ← beq_nat_refl.
  - apply neq_comm in n.
    rewrite beq_nat_false'.
    assert (a:= inb_tail y x xs n e ).
    destruct (natlist_reflect y xs).
    + easy.
    + easy.
    + now apply neq_comm.
  - assert (a1:= inb_false1 y x xs e).
    assert (a2:= inb_false2 y x xs e).
    apply neq_comm in a1.
    destruct (natlist_reflect y xs).
    + easy.
    + now rewrite beq_nat_false'.
Qed.

Lemma values_of_01sub_list : ∀ l x gamma ,
    ls_01sub_list l gamma →
```

```
    gamma x = T0 ∨ gamma x = T1.
Proof.
  induction l as [| y ys IH].
  - intros x gamma H.
    inv H; left; reflexivity.
  - intros x gamma H.
    unfold update_sub.
    inversion H; subst.
    + decide (y = x).
      – rewrite e; now left.
      – assert (a:= apply_updated_sub_off_var sigma y x T0 n).
        rewrite a.
        apply IH; easy.
    + decide (y = x).
      – rewrite e; now right.
      – assert (a:= apply_updated_sub_off_var sigma y x T1 n).
        rewrite a.
        apply IH; easy.
Qed.

Lemma values_of_01sub : ∀ gamma ,
    ls_01sub gamma →
    ∀ x, gamma x = T0 ∨ gamma x = T1.
Proof.
  unfold ls_01sub.
  intros gamma H x.
  destruct H as [l H1].
  now apply (values_of_01sub_list l).
Qed.

Lemma truncate_preserves_01 : ∀ l gamma,
    ls_01sub gamma →
    ls_01sub_list l (truncate0 l gamma).
Proof.
  induction l as [| x xs IH].
  - intros gamma H. apply ls_01sub_listNil .
  - intros gamma H.
    assert (a:= IH gamma H).
    assert (b: truncate0 (x :: xs) gamma =
                update_sub (truncate0 xs gamma) x (gamma x)).
    apply truncate_update.
    assert (c:= values_of_01sub gamma H x).
    destruct c as [c0 | c1].
    + rewrite b.
```

```
    rewrite c0.
    now apply ls_01sub_list0.
  + rewrite b.
    rewrite c1.
    now apply ls_01sub_list1.
Qed.
```

6.2.2 Updating by T0 or T1 preserves 01-ness

```
Lemma update_preserves_01 (sigma : sub) (x : var) (t : bterm) :
  ls_01sub sigma →
  ( (t = T0) ∨ (t = T1) ) →
  ls_01sub (update_sub sigma x t).
Proof.
  unfold ls_01sub.
  intros H H01.
  destruct H as [l HH].
  destruct H01 as [H0 | H1].
  - rewrite H0. ∃ (x::l). now apply ls_01sub_list0.
  - rewrite H1. ∃ (x::l). now apply ls_01sub_list1.
Qed.

Lemma ls_01sub_gives_Ground (sigma : sub) (t : bterm) :
  ls_01sub sigma →
  Ground_bterm (sigma ' t).
Proof.
  intros H.
  assert (a:= values_of_01sub sigma H).
  induction t as [ | | y | t1 IH1 t2 IH2 | t1 IH1 t2 IH2 ]; simpl; auto.
  assert (b:= a y).
  destruct b as [b0 | b1]; simpl; auto.
  rewrite b0; auto.
  rewrite b1; auto.
Qed.
Hint Resolve Is_01sub_gives_Ground.

Lemma in_all01subs sigma x xs :
  sigma el (all_01subs_list (x::xs)) →
  sigma el (map (fun s ⇒ update_sub s x T0) (all_01subs_list xs)) ∨
  sigma el (map (fun s ⇒ update_sub s x T1) (all_01subs_list xs)).
Proof.
  intros.
  apply in_app_or.
  firstorder.
```

```
Qed.

Lemma in_all01subs0 (sigma: sub) x xs:
  sigma el (map (fun s ⇒ update_sub s x T0) (all_01subs_list xs)) ↔
  ∃ sigma', update_sub sigma' x T0 = sigma ∧
                    sigma' el (all_01subs_list xs).
Proof.
  apply in_map_iff.
Qed.

Lemma in_all01subs1 (sigma: sub) x xs:
  sigma el (map (fun s ⇒ update_sub s x T1) (all_01subs_list xs)) ↔
  ∃ sigma', update_sub sigma' x T1 = sigma ∧
                    sigma' el (all_01subs_list xs).
Proof.
  apply in_map_iff.
Qed.
```

in fact iff is true as well

```
Lemma in_all01subs_if x xs (sigma: sub) :
  sigma el all_01subs_list (x::xs)
  →
  (∃ sigma', update_sub sigma' x T0 = sigma ∧
                    sigma' el (all_01subs_list xs))
  ∨
  (∃ sigma', update_sub sigma' x T1 = sigma ∧
                    sigma' el (all_01subs_list xs)).
Proof.
  intros H.
  assert (a:= in_all01subs sigma x xs H).
  destruct a as [a0 | a1].
  - left. now apply in_all01subs0.
  - right. now apply in_all01subs1.
Qed.

Lemma all01_All01 : ∀ vars sigma,
    sigma el (all_01subs_list vars) →
    Is_01sub_list vars sigma.
Proof.
  intros vars sigma.
  generalize dependent sigma.
  induction vars as [|x xs IH].
  - intros sigma H.
    unfold all_01subs_list in H.
    apply in_sing in H.
```

```
    rewrite H. auto.
  - intros sigma H.
    assert (a:= in_all01subs_if x xs sigma H).
    destruct a as [a1 | a2].
    + destruct a1 as [sigma'].
      destruct H0.
      unfold update_sub in H0.
      rewrite ← H0.
      apply ls_01sub_list0.
      now apply IH.
    + destruct a2 as [sigma'].
      destruct H0.
      unfold update_sub in H0.
      rewrite ← H0.
      apply ls_01sub_list1.
      now apply IH.
Qed.

Lemma all_01sub_gives_Ground (sigma : sub) (t : bterm) (l : list var) :
  sigma el all_01subs_list l →
  Ground_bterm (sigma ' t).
Proof.
  intros H.
  apply all01_All01 in H.
  apply ls_01sub_gives_Ground.
  unfold ls_01sub.
  now ∃ l.
Qed.
Hint Resolve all_01sub_gives_Ground.

Lemma All01_all01 vars sigma :
  ls_01sub_list vars sigma →
  sigma el all_01subs_list vars.
Proof.
  intros H.
  induction H.
  - unfold all_01subs_list.
      easy.
  - simpl.
    apply el_app_L.
    assert (a:= in_map (fun s : sub ⇒ update_sub s x T0)
                       (all_01subs_list l)
                       sigma
                       IHls_01sub_list).
```

```
    now cbn in a.
  - simpl.
    apply el_app_R.
    assert (a:= in_map (fun s : sub ⇒ update_sub s x T1)
                       (all_01subs_list l)
                       sigma
                       IHIs_01sub_list).
    now cbn in a.
Qed.

Lemma truncate0_ls_01 :
  ∀ (ltrunc: list var) (gamma : sub),
    ls_01sub gamma →
    ls_01sub_list ltrunc (truncate0 ltrunc gamma).
Proof.
  intros ltrunc gamma H.
  now apply (truncate_preserves_01 ltrunc gamma H).
Qed.

Lemma All01_all01_bterm gamma t :
  ls_01sub gamma →
  ∃ gamma',
    gamma' el (all_01subs_list (vars_bterm t)) ∧
    gamma' 't = gamma ' t.
Proof.
  intros H.
  remember H as Hgamma.
  unfold ls_01sub in H.
  destruct H as [lgamma H'].
  assert (a:= truncate0_suffices t gamma).
  assert (b:= truncate0_ls_01 (vars_bterm t) gamma Hgamma).
  assert (c:= All01_all01 (vars_bterm t) (truncate0 (vars_bterm t) gamma) b).
  ∃ (truncate0 (vars_bterm t) gamma).
  split. easy. easy.
Qed.
```

Chapter 7

Library BooleanRings.EvalGround

```
Require Export Omega.
Require Export List.
Export ListNotations.
Require Export Setoid.
Require Export Morphisms.
Require Export Relation_Definitions.
Require Export Ring.
Require Export Basics.
Require Export Utilities.
Require Export BTermsUtilities.
Require Export Substitutions.
Require Export Coq.Program.Equality.
```

7.1 Evaluating (ground) bterms

When t is ground, evaluates t to T0 or to T1. Not complete when t is not ground

```
Fixpoint eval_ground (t: bterm) : bterm :=
  match t with
  | T0 ⇒ T0
  | T1 ⇒ T1
  | (V _) ⇒ t
  | A u v ⇒ match (eval_ground u) , (eval_ground v) with
              | T0 , T0 ⇒ T0
              | T0 , T1 ⇒ T1
              | T1 , T0 ⇒ T1
              | T1 , T1 ⇒ T0
              | _ , _ ⇒ t
              end
```

```
  | M u v ⇒ match (eval_ground u) , (eval_ground v) with
             | T0 , T0 ⇒ T0
             | T0 , T1 ⇒ T0
             | T1 , T0 ⇒ T0
             | T1 , T1 ⇒ T1
             | _ , _ ⇒ t
             end
  end.
```

7.1.1 eqv is a congruence wrt eval_ground

```
Lemma eval_ground_eqv t :
  eval_ground t == t.
Proof.
  induction t as [ | | v | t1 IH1 t2 IH2| t1 IH1 t2 IH2]; simpl; auto;
    destruct (eval_ground t1); destruct (eval_ground t2);
      rewrite ← IH1; rewrite ← IH2; auto.
Qed.
Hint Resolve eval_ground_eqv.

Lemma eval_ground_compat t1 t2 :
  t1 == t2 → (eval_ground t1) == (eval_ground t2).
Proof.
  intros H.
  assert (a1:= eval_ground_eqv t1).
  assert (a2:= eval_ground_eqv t2).
  apply eqv_trans with t1. easy.
  apply eqv_trans with t2. easy. easy.
Qed.
Hint Resolve eval_ground_compat.

Add Parametric Morphism : eval_ground with
      signature eqv ==> eqv as eval_ground_mor.
  exact eval_ground_compat.
Qed.

Lemma eval_ground_complete (t: bterm) :
  (ground_bterm t = true) →
  (eval_ground t = T0) ∨ (eval_ground t = T1).
Proof.
  intros H.
  induction t as [| | v | t1 IH1 t2 IH2 | t1 IH1 t2 IH2]; simpl; auto.
  - inversion H.
  - apply ground_bterm_A_elim in H; destruct H;
      assert (a:= IH1 H);
```

```
      assert (b:= IH2 H0);
      destruct a;
      destruct b;
      rewrite H1;
      rewrite H2;
      firstorder.
  - apply ground_bterm_M_elim in H; destruct H;
      assert (a:= IH1 H);
      assert (b:= IH2 H0);
      destruct a;
      destruct b;
      rewrite H1;
      rewrite H2;
      firstorder.
Qed.
Hint Resolve eval_ground_complete.

Lemma eval_Ground_complete (t: bterm) :
  Ground_bterm t →
  (eval_ground t = T0) ∨ (eval_ground t = T1).
Proof.
  intros H.
  apply Ground_is_ground in H.
  now apply eval_ground_complete.
Qed.
Hint Resolve eval_Ground_complete.

Lemma Ground_T0_or_T1 (t: bterm) :
  Ground_bterm t →
  (t == T0) ∨ ( t == T1).
Proof.
  intros H.
  assert (A:= eval_Ground_complete t H).
  assert (B:= (eval_ground_eqv t)).
  destruct A as [l | r].
  - left.
    rewrite l in B.
    now apply eqv_sym.
  - right.
    rewrite r in B.
    now apply eqv_sym.
Qed.
Hint Resolve Ground_T0_or_T1.

Lemma not_T0_then_T1 (t: bterm) :
```

```
  Ground_bterm t →
  ¬ (t == T0) → ( t == T1).
Proof.
  assert (a:= Ground_T0_or_T1 t).
  tauto.
Qed.

Lemma not_T1_then_T0 (t: bterm) :
  Ground_bterm t →
  ¬ (t == T1) → (t == T0).
Proof.
  assert (a:= Ground_T0_or_T1 t).
  tauto.
Qed.

Lemma eval_ground_exact_T0 t :
  eval_ground t = T0 →
  t == T0.
Proof.
  intros H0.
  rewrite ← H0.
  symmetry; apply eval_ground_eqv.
Qed.
Hint Resolve eval_ground_exact_T0.

Lemma eval_ground_exact_T1 t :
  eval_ground t = T1 →
  t == T1.
Proof.
  intros H1.
  rewrite ← H1.
  symmetry; apply eval_ground_eqv.
Qed.
Hint Resolve eval_ground_exact_T1.
```

Chapter 8

Library BooleanRings.B2

```
From Coq Require Export Bool.
Require Export Omega.
Require Export List.
Export ListNotations.
From Coq Require Export Lists.ListSet.
Require Export Basics.
Require Export Setoid.
Require Export Morphisms.
Require Export Relation_Definitions.
Require Export Logic.FunctionalExtensionality.
Require Export Utilities.
Require Export TheoryDef.
Require Export BTermsUtilities.
Require Export EvalGround.
Require Export Substitutions.
Require Export Substitutions01.
```

8.1 Show that bool is a model of the theory.

```
Definition BF (t1 t2 : bterm) : Prop :=
  ∀ (sigma: sub),
    ls_01sub sigma →
    eval_ground (sigma ' t1) = eval_ground (sigma ' t2).

Lemma B2_helper (sigma : sub) (t : bterm) :
  ls_01sub sigma →
  eval_ground (sigma ' t) = T0 ∨
  eval_ground (sigma ' t) = T1.
Proof.
```

```
  intros H.
  apply eval_ground_complete.
  apply Ground_is_ground.
  now apply Is_01sub_gives_Ground.
Qed.
```

8.1.1 Structural rules

```
Lemma BF_is_refl (x : bterm) :
  BF x x.
Proof.
  easy.
Qed.

Lemma BF_is_sym (x y : bterm) :
  BF x y → BF y x.
Proof.
  unfold BF.
  intros H sigma H1.
  apply symmetry. firstorder.
Qed.

Lemma BF_is_trans (x y z : bterm) :
  BF x y → BF y z → BF x z .
  unfold BF.
  intros H1 H2 sig H3.
  assert (H11 := H1 sig H3).
  assert (H21 := H2 sig H3).
  now rewrite H11.
Qed.
```

This cries out for some automation

```
Lemma BF_is_A_compat (x y x' y' : bterm) :
  BF x x' → BF y y' → BF (x +' y) (x' +' y').
Proof.
  unfold BF.
  intros H1 H2 sig H.
  assert (H11 := H1 sig H).
  assert (H21 := H2 sig H).

  simpl.

  assert (Hxvals := B2_helper sig x H).
  assert (Hx'vals := B2_helper sig x' H).
  assert (Hyvals := B2_helper sig y H).
  assert (Hy'vals := B2_helper sig y' H).
```

```
  destruct Hxvals as [Dx0 | Dx1].
  - rewrite Dx0.
    destruct Hyvals as [Dy0 | Dy1].
    + rewrite Dy0.
      destruct Hx'vals as [Dx'0 | Dx'1].
      – rewrite Dx'0.
          destruct Hy'vals as [Dy'0 | Dy'1].
          ++ now rewrite Dy'0.
          ++ congruence.
      – rewrite Dx'1.
          destruct Hy'vals as [Dy'0 | Dy'1].
          ++ congruence.          ++ congruence.
    + rewrite Dy1.
      destruct Hx'vals as [Dx'0 | Dx'1].
      – rewrite Dx'0.
          destruct Hy'vals as [Dy'0 | Dy'1].
          ++ congruence.          ++ now rewrite Dy'1.
      – rewrite Dx'1.
          destruct Hy'vals as [Dy'0 | Dy'1].
          ++ congruence.          ++ congruence.
  - rewrite Dx1.
    destruct Hyvals as [Dy0 | Dy1].
    + rewrite Dy0.
      destruct Hx'vals as [Dx'0 | Dx'1].
      – rewrite Dx'0.
          destruct Hy'vals as [Dy'0 | Dy'1].
          ++ congruence.          ++ congruence.
      – rewrite Dx'1.
          destruct Hy'vals as [Dy'0 | Dy'1].
          ++ now rewrite Dy'0.
          ++ congruence.
    + rewrite Dy1.
      destruct Hx'vals as [Dx'0 | Dx'1].
      – rewrite Dx'0.
          destruct Hy'vals as [Dy'0 | Dy'1].
          ++ congruence.          ++ congruence.
      – rewrite Dx'1.
          destruct Hy'vals as [Dy'0 | Dy'1].
          ++ congruence.          ++ now rewrite Dy'1.
Qed.

Lemma BF_is_M_compat (x y x' y' : bterm) :
```

```
  BF x x' → BF y y' → BF (x *' y) (x' *' y').
Proof.
  unfold BF.
  intros H1 H2 sig H.
  assert (H11 := H1 sig H).
  assert (H21 := H2 sig H).
  simpl.
  assert (Hxvals := B2_helper sig x H).
  assert (Hx'vals := B2_helper sig x' H).
  assert (Hyvals := B2_helper sig y H).
  assert (Hy'vals := B2_helper sig y' H).
  destruct Hxvals as [Dx0 | Dx1].
  - rewrite Dx0.
    destruct Hyvals as [Dy0 | Dy1].
    + rewrite Dy0.
      destruct Hx'vals as [Dx'0 | Dx'1].
      – rewrite Dx'0.
        destruct Hy'vals as [Dy'0 | Dy'1].
        ++ now rewrite Dy'0.
        ++ congruence.
      – rewrite Dx'1.
        destruct Hy'vals as [Dy'0 | Dy'1].
        ++ congruence.          ++ congruence.
    + rewrite Dy1.
      destruct Hx'vals as [Dx'0 | Dx'1].
      – rewrite Dx'0.
        destruct Hy'vals as [Dy'0 | Dy'1].
        ++ congruence.          ++ now rewrite Dy'1.
      – rewrite Dx'1.
        destruct Hy'vals as [Dy'0 | Dy'1].
        ++ congruence.          ++ congruence.
  - rewrite Dx1.
    destruct Hyvals as [Dy0 | Dy1].
    + rewrite Dy0.
      destruct Hx'vals as [Dx'0 | Dx'1].
      – rewrite Dx'0.
        destruct Hy'vals as [Dy'0 | Dy'1].
        ++ congruence.          ++ congruence.
      – rewrite Dx'1.
        destruct Hy'vals as [Dy'0 | Dy'1].
```

```
        ++ now rewrite Dy'0.
        ++ congruence.
    + rewrite Dy1.
      destruct Hx'vals as [Dx'0 | Dx'1].
      - rewrite Dx'0.
        destruct Hy'vals as [Dy'0 | Dy'1].
        ++ congruence.          ++ congruence.
      - rewrite Dx'1.
        destruct Hy'vals as [Dy'0 | Dy'1].
        ++ congruence.          ++ now rewrite Dy'1.
Qed.
```

8.1.2 The axioms

All the proofs with same number of variables are the same! Just introduce them, establish that they are each either T0 or T1, and compute

One Variable

```
Lemma invA_BF (x : bterm) : BF (x +' x ) T0 .
Proof.
  unfold BF.
  intros sigma H.
  simpl.
  assert (Hxvals:= B2_helper sigma x H).
  destruct Hxvals as [Dx0 | Dx1].
  - now rewrite Dx0.
  - now rewrite Dx1.
Qed.
Lemma idA_BF (x : bterm) : BF (T0 +' x ) x .
Proof.
  unfold BF.
  intros sigma H.
  simpl.
  assert (Hxvals:= B2_helper sigma x H).
  destruct Hxvals as [Dx0 | Dx1].
  - now rewrite Dx0.
  - now rewrite Dx1.
Qed.
Lemma ord2M_BF (x : bterm) : BF (x *' x ) x .
Proof.
```

```
  unfold BF.
  intros sigma H.
  simpl.
  assert (Hxvals:= B2_helper sigma x H).
  destruct Hxvals as [Dx0 | Dx1].
  - now rewrite Dx0.
  - now rewrite Dx1.
Qed.
Lemma idM_BF (x : bterm) : BF (T1 *' x ) x .
Proof.
  unfold BF.
  intros sigma H.
  simpl.
  assert (Hxvals:= B2_helper sigma x H).
  destruct Hxvals as [Dx0 | Dx1].
  - now rewrite Dx0.
  - now rewrite Dx1.
Qed.
```

Two Variables

```
Lemma comA_BF (x y : bterm) : BF (x +' y) (y +' x) .
Proof.
  unfold BF.
  intros sigma H.
  simpl.
  assert (Hxvals:= B2_helper sigma x H).
  assert (Hyvals:= B2_helper sigma y H).
  destruct Hxvals as [Dx0 | Dx1].
  - rewrite Dx0.
    destruct Hyvals as [Dy0 | Dy1].
    + now rewrite Dy0.
    + now rewrite Dy1.
  - rewrite Dx1.
    destruct Hyvals as [Dy0 | Dy1].
    + now rewrite Dy0.
    + now rewrite Dy1.
Qed.
```

Three Variables

```
Lemma assocA_BF (x y z : bterm) : BF ((x +' y) +' z) (x +' (y +' z)).
```

```
Proof.
  unfold BF.
  intros sig H.
  simpl.
  assert (Hxvals := B2_helper sig x H).
  assert (Hyvals := B2_helper sig y H).
  assert (Hzvals := B2_helper sig z H).

  simpl.
  destruct Hxvals as [Dx0 | Dx1].
  - rewrite Dx0.
    destruct Hyvals as [Dy0 | Dy1].
    + rewrite Dy0.
      destruct Hzvals as [Dz0 | Dz1].
      × now rewrite Dz0.
      × now rewrite Dz1.

    + rewrite Dy1.
      destruct Hzvals as [Dz0 | Dz1].
      × now rewrite Dz0.
      × now rewrite Dz1.

  - rewrite Dx1.
    destruct Hyvals as [Dy0 | Dy1].
    + rewrite Dy0.
      destruct Hzvals as [Dz0 | Dz1].
      × now rewrite Dz0.
      × now rewrite Dz1.

    + rewrite Dy1.
      destruct Hzvals as [Dz0 | Dz1].
      × now rewrite Dz0.
      × now rewrite Dz1.
Qed.

Lemma assocM_BF (x y z : bterm) :
  BF ((x *' y) *' z ) ( x *' (y *' z)) .
Proof.
  unfold BF.
  intros sig H.
  simpl.
  assert (Hxvals := B2_helper sig x H).
  assert (Hyvals := B2_helper sig y H).
  assert (Hzvals := B2_helper sig z H).

  simpl.
  destruct Hxvals as [Dx0 | Dx1].
```

```
  - rewrite Dx0.
    destruct Hyvals as [Dy0 | Dy1].
    + rewrite Dy0.
      destruct Hzvals as [Dz0 | Dz1].
      × now rewrite Dz0.
      × now rewrite Dz1.
    + rewrite Dy1.
      destruct Hzvals as [Dz0 | Dz1].
      × now rewrite Dz0.
      × now rewrite Dz1.
  - rewrite Dx1.
    destruct Hyvals as [Dy0 | Dy1].
    + rewrite Dy0.
      destruct Hzvals as [Dz0 | Dz1].
      × now rewrite Dz0.
      × now rewrite Dz1.
    + rewrite Dy1.
      destruct Hzvals as [Dz0 | Dz1].
      × now rewrite Dz0.
      × now rewrite Dz1.
Qed.

Lemma dist_BF (x y z : bterm) :
  BF ((y +' z) *' x ) ( (y *' x) +' (z *' x)) .
Proof.
  unfold BF.
  intros sig H.
  simpl.
  assert (Hxvals := B2_helper sig x H).
  assert (Hyvals := B2_helper sig y H).
  assert (Hzvals := B2_helper sig z H).
  simpl.
  destruct Hxvals as [Dx0 | Dx1].
  - rewrite Dx0.
    destruct Hyvals as [Dy0 | Dy1].
    + rewrite Dy0.
      destruct Hzvals as [Dz0 | Dz1].
      × now rewrite Dz0.
      × now rewrite Dz1.
    + rewrite Dy1.
      destruct Hzvals as [Dz0 | Dz1].
      × now rewrite Dz0.
```

```
      × now rewrite Dz1.
  - rewrite Dx1.
    destruct Hyvals as [Dy0 | Dy1].
    + rewrite Dy0.
      destruct Hzvals as [Dz0 | Dz1].
      × now rewrite Dz0.
      × now rewrite Dz1.
    + rewrite Dy1.
      destruct Hzvals as [Dz0 | Dz1].
      × now rewrite Dz0.
      × now rewrite Dz1.
Qed.

Lemma dist'_BF (x y z : bterm) :
  BF (x *' (y +' z)) ( (x *' y) +' (x *' z)) .
Proof.
  unfold BF.
  intros sig H.
  simpl.
  assert (Hxvals := B2_helper sig x H).
  assert (Hyvals := B2_helper sig y H).
  assert (Hzvals := B2_helper sig z H).
  simpl.
  destruct Hxvals as [Dx0 | Dx1].
  - rewrite Dx0.
    destruct Hyvals as [Dy0 | Dy1].
    + rewrite Dy0.
      destruct Hzvals as [Dz0 | Dz1].
      × now rewrite Dz0.
      × now rewrite Dz1.
    + rewrite Dy1.
      destruct Hzvals as [Dz0 | Dz1].
      × now rewrite Dz0.
      × now rewrite Dz1.
  - rewrite Dx1.
    destruct Hyvals as [Dy0 | Dy1].
    + rewrite Dy0.
      destruct Hzvals as [Dz0 | Dz1].
      × now rewrite Dz0.
      × now rewrite Dz1.
    + rewrite Dy1.
      destruct Hzvals as [Dz0 | Dz1].
```

```
    × now rewrite Dz0.
    × now rewrite Dz1.
Qed.

Hint Resolve
    assocA_BF invA_BF idA_BF comA_BF assocM_BF ord2M_BF idM_BF
    dist_BF dist'_BF BF_is_refl BF_is_sym BF_is_trans BF_is_A_compat
    BF_is_M_compat .

Lemma BF_is_model (t1 t2 : bterm) :
  t1 == t2 → BF t1 t2.
Proof.
  intros H.
  induction H; simpl; auto.
  now apply BF_is_trans with y.
Qed.

Lemma T0_not_BF_T1 :
  ¬ BF T0 T1.
Proof.
  intros H.
  unfold BF in H.
  assert (a:= H const0sub const0_is_01sub).
  easy.
Qed.

Lemma T0_neqv_T1:
  ¬ (T0 == T1).
Proof.
  intros H.
  apply T0_not_BF_T1.
  now apply BF_is_model.
Qed.
Hint Resolve T0_neqv_T1.

Lemma T1_neqv_T0 :
  ¬ (T1 == T0).
Proof.
  intros H. apply eqv_sym in H. now apply T0_neqv_T1.
Qed.
Hint Resolve T1_neqv_T0.

Lemma eval_ground_eqv_T0 t :
  Ground_bterm t →
  t == T0 →
  eval_ground t = T0 .
Proof.
```

```
  intros H0 H1.
  assert (H2 := Ground_is_ground t H0).
  assert (H3 := eval_ground_complete t H2).
  inversion H3.
  - easy.
  - assert (H4 := eval_ground_exact_T1 t H).
    rewrite H1 in H4.
    symmetry in H4.
    now apply T1_neqv_T0 in H4.
Qed.

Lemma eval_ground_eqv_T1 t :
  Ground_bterm t →
  t == T1 →
  eval_ground t = T1 .
Proof.
  intros H0 H1.
  assert (H2 := Ground_is_ground t H0).
  assert (H3 := eval_ground_complete t H2).
  inversion H3.
  - assert (H4 := eval_ground_exact_T0 t H).
    rewrite H1 in H4.
    symmetry in H4.
    now apply T0_neqv_T1 in H4.
  - easy.
Qed.

Lemma ground_T0_iff t :
  Ground_bterm t →
  t == T0 ↔ ¬ t == T1.
Proof.
  intros H1. split.
  - intros H2 H3.
    rewrite H2 in H3. auto.
  - intros H4.
    assert (A:= eval_Ground_complete t H1).
    destruct A as [L | R].
    + now apply eval_ground_exact_T0.
    + rewrite (eval_ground_exact_T1 t R) in H4.
      assert (A:= eqv_ref T1).
      tauto.
Qed.
```

Once again: conceptually just use ground_T0_iff for t +' 1. But more trouble to prove

it that way than to just cut-and-paste that proof with obvious changes

```
Lemma ground_T1_iff t :
  Ground_bterm t →
  t == T1 ↔ ¬ t == T0.
Proof.
  intros H1. split.
  - intros H2 H3.
    rewrite H2 in H3. auto.
  - intros H4.
    assert (A:= eval_Ground_complete t H1).
    destruct A as [L | R].
    + rewrite (eval_ground_exact_T0 t L) in H4.
      assert (A:= eqv_ref T0).
      tauto.
    + now apply eval_ground_exact_T1.
Qed.
```

Chapter 9

Library BooleanRings.Factoring

```
Require Export Omega.
Require Export List.
Export ListNotations.
Export Setoid.

Require Export Utilities.
Require Export TheoryDef.
Require Export BTermsUtilities.
Require Export Substitutions.
```

9.1 Defining quotient and remainder

We actually do not use this definition now; we leave it here as motivation/explanation for the direct definitions of quotient and remainder below

```
Fixpoint quotRem (t: bterm) (x: var) : (bterm × bterm) :=
  match t with
  | T0 ⇒ (T0, T0)
  | T1 ⇒ (T0, T1)
  | V v ⇒ if Nat.eqb x v then (T1, T0) else (T0, t)
  | A t1 t2 ⇒
    let (q1, r1) := (quotRem t1 x) in
    let (q2, r2) := (quotRem t2 x) in
    (q1 +' q2, r1 +' r2)
  | M t1 t2 ⇒
    let (q1, r1) := (quotRem t1 x) in
    let (q2, r2) := (quotRem t2 x) in
    (q1 *' q2 +' q1 *' r2 +' q2 *' r1, r1 *' r2)
  end.
Hint Unfold quotRem.
```

These direct definitions make it much easier to prove things about quotient and remainder (for example the variables they contain)

These are just the "projections" of the definition in quotRem. Fortunately, although the defn of quotient is both recursive and uses remainder, the definition of remainder does not use quotient. So we can do that first.

```
Fixpoint remainder (t: bterm) (x: var) : bterm :=
  match t with
  | T0 ⇒ T0
  | T1 ⇒ T1
  | V v ⇒ if eqb_nat x v then T0 else t
  | A t1 t2 ⇒
    let r1 := (remainder t1 x) in
    let r2 := (remainder t2 x) in
    (r1 +' r2)
  | M t1 t2 ⇒
    let r1 := (remainder t1 x) in
    let r2 := (remainder t2 x) in
    (r1 *' r2)
  end.
Hint Unfold remainder.
```

```
Fixpoint quotient (t: bterm) (x: var) : bterm :=
  match t with
  | T0 ⇒ T0
  | T1 ⇒ T0
  | V v ⇒ if eqb_nat x v then T1 else T0
  | A t1 t2 ⇒
    let q1 := (quotient t1 x) in
    let q2 := (quotient t2 x) in
    (q1 +' q2)
  | M t1 t2 ⇒
    let q1 := (quotient t1 x) in
    let q2 := (quotient t2 x) in
    (q1 *' q2 +' q1 *' (remainder t2 x) +' q2 *' (remainder t1 x))
  end.
Hint Unfold quotient.
```

Write t as x * q + r

```
Definition factor (t: bterm) (x: var) : bterm :=
  ( (V x) *' (quotient t x) +' (remainder t x)).
```

9.2 Reasoning About Quotient and Remainder

```
Lemma eqb_nat_reflect : ∀ (x y : nat), reflect (x = y) (x =? y).
Proof.
  intros x y.
  apply iff_reflect. symmetry. apply beq_nat_true_iff.
Qed.

Lemma vars_remainder_rem (t: bterm) (x: var) :
  vars_bterm (remainder t x) «=
  rem (vars_bterm t) x.
Proof.
  intros.
  induction t as [ | | v | t1 IH1 t2 IH2 | t1 IH1 t2 IH2 ];
    simpl; auto.
  - destruct (eqb_nat_reflect x v).
    + rewrite e. simpl. rewrite ← beq_nat_refl.
      now rewrite eq_if_neq_dec.
    + rewrite beq_nat_false'.
      apply neq_comm in n. rewrite neq_if_neq_dec; easy. easy.
  - rewrite IH1; rewrite IH2.
    now rewrite rem_app_hom.
  - rewrite IH1; rewrite IH2.
    now rewrite rem_app_hom.
Qed.
```

9.2.1 Neither the quotient nor the remainder contain x

```
Lemma not_in_vars_remainder (t: bterm) (x: var) :
  ¬ (x el (vars_bterm (remainder t x))).
Proof.
  assert (A:= vars_remainder_rem t x).
  now assert (A2:= not_in_rem_incl
                   x
                   (vars_bterm (remainder t x))
                   (vars_bterm t) A).
Qed.

Lemma vars_quotient_rem t x :
  vars_bterm (quotient t x) «=
             rem (vars_bterm t) x.
Proof.
  intros.
```

```
  induction t as [ | | v | t1 IH1 t2 IH2 | t1 IH1 t2 IH2 ];
    simpl; auto.
  - decide (x=v).
    + rewrite e. simpl. rewrite ← beq_nat_refl.
      now rewrite eq_if_neq_dec.
    + assert (a:= neq_comm x v n ).
      rewrite neq_if_neq_dec.
      rewrite beq_nat_false'.
      easy. easy. easy.
  - rewrite IH1; rewrite IH2.
    now rewrite rem_app_hom.
  - assert (A1:= vars_remainder_rem t1 x).
    assert (A2:= vars_remainder_rem t2 x).
    rewrite IH1; rewrite IH2.
    repeat rewrite app_assoc_reverse.
    rewrite ← rem_app_hom; apply incl_app; auto.
    apply incl_app; auto.
Qed.

Lemma not_in_vars_quotient (t: bterm) (x: var) :
  ¬ (x el (vars_bterm (quotient t x))).
Proof.
  assert (A:= vars_quotient_rem t x).
  now assert (A2:= not_in_rem_incl
                   x
                   (vars_bterm (quotient t x))
                   (vars_bterm t) A).
Qed.
```

9.2.2 Correctness of the factoring

The addition case of factoring is correct

```
Lemma quotRem_A: ∀ t1 t2 x q1 r1 q2 r2,
    t1 == ( (V x) *' q1) +' r1 →
    t2 == ( (V x) *' q2) +' r2 →
    t1 +' t2 == ( (V x) *' (q1 +' q2) +' (r1 +' r2) ) .
Proof.
  intros t1 t2 x q1 r1 q2 r2 H1 H2.
  rewrite H1. rewrite H2.
  bsimp.
Qed.
```

The multiplication case of factoring is correct

```
Lemma quotRem_M: ∀ t1 t2 x q1 r1 q2 r2,
    t1 == ( (V x) *' q1) +' r1 →
    t2 == ( (V x) *' q2) +' r2 →
    t1 *' t2 == ( (V x) *' ((q1 *' q2) +' (q1 *' r2) +' (q2 *' r1))
                  +' (r1 *' r2) ) .
Proof.
  intros t1 t2 x q1 r1 q2 r2 H1 H2.
  rewrite H1. rewrite H2.
  bsimp.
Qed.
```

Factoring t by any variable preserves eqv

```
Lemma factor_correct :
  ∀ (t: bterm) (x: var), t == factor t x .
Proof.
  induction t as [ | | v | t1 IH1 t2 IH2 | t1 IH1 t2 IH2 ];
    unfold factor; unfold quotient; unfold remainder; bsimp.
  - intros; simpl; bsimp.
  - intros; simpl; bsimp.
  - intros.
    destruct (eqb_nat_reflect x v).
    + rewrite e; bsimp.
       rewrite ← beq_nat_refl.
       bsimp.
    + bsimp. rewrite beq_nat_false'. bsimp. easy.
  - intros. unfold factor in *.
    assert (H1 := IH1 x).
    assert (H2 := IH2 x).
    (assert (a:= quotRem_A
                   t1 t2 x
                   (quotient t1 x) (remainder t1 x)
                   (quotient t2 x) (remainder t2 x) H1 H2)).
    simpl.
    destruct (quotRem t1 x).
    destruct (quotRem t2 x).
    rewrite a.
    simpl.
    bsimp.
  - intros. unfold factor in *.
```

```
    assert (H1 := IH1 x).
    assert (H2 := IH2 x).
    (assert (a:= quotRem_M
                   t1 t2 x
                   (quotient t1 x) (remainder t1 x)
                   (quotient t2 x) (remainder t2 x) H1 H2)).
    simpl.
    destruct (quotRem t1 x).
    destruct (quotRem t2 x).
    rewrite a.
    simpl.
    bsimp.
Qed.

Lemma factor_neq_T0 s x :
  ¬ (s == T0) →
  ¬ (remainder s x) == T0 ∨
  ¬ (quotient s x) == T0 .
Proof.
  apply classical_contra.
  intros.
  apply not_or_and in H.
  destruct H as [Rem0 Quo0].
  apply NNPP in Rem0. apply NNPP in Quo0.
  cut (s == T0).
  - tauto.
  - assert (Factor:= factor_correct s x).
    rewrite Factor.
    unfold factor.
    rewrite Rem0. rewrite Quo0.
    bsimp.
Qed.

Lemma factor_neq_T0_strong s x :
  ¬ (s == T0) →
  ¬ (remainder s x) == T0 ∨
  (remainder s x) == T0 ∧ ¬ (quotient s x) == T0 .
Proof.
  intros H.
  assert (A:= factor_neq_T0 s x).
  assert (B:= A H).
  apply strong_disjunct_L in B.
  firstorder.
  apply NNPP in H0.
```

  now `right`.
`Qed`.

Chapter 10

Library Boolean-Rings.EquivalenceAndSolvability

```
Require Export Sumbool.
Require Export DecBool.

Require Export Omega.
Require Export List.
Export ListNotations.
Require Export Setoid.
Require Export Morphisms.
Require Export Relation_Definitions.
Require Export Ring.
Require Export Basics.
Require Export Classical_Prop.
Require Export Classical_Pred_Type.

Require Export Utilities.
Require Export BTermsUtilities.
Require Export Substitutions.
Require Export Substitutions01.
Require Export EvalGround.
Require Export Factoring.
Require Export B2.
```

10.1 Characterizing T0-equivalence

10.1.1 Overview

- Key notion: a 01_sub, which maps every var to 0 or to 1.
- Main Theorem is

`t == T0 iff forall 01 substitutions gamma, gamma t == T0`

The proof of (1) uses the factoring idea, that for any t and variable x, we can find q and r such that t == qx + r. This is also used in the the Variable Elimination algorithm.

So, for the hard direction: suppose (t == T0) fails. We will build a ground subst sigma such that sigma t == T1.

Induction on number of variables in t.

If t is ground then t evaluates to either T0 or T1, ok, take sigma = id.

Else write t == qx+r for some q,r not involving x. At least one of q, r is not eqv T0.

If r is not T0, then by induction we have sigma' with sigma' r = T1. Then take sigma to be sigma' + x |-> 0

If r == T0 and q is not == T0, then by induction we have sigma' with sigma' q == T1. Then take sigma be sigma' + x |-> 1

- Given that theorem, we get

  `t == T1 iff forall 01_subs gamma, gamma t == T1`

  Proof. We can just apply previous theorem to the bterm (t + T1). But this proof technique is annoyingly hard to implement so here we just essentially replay the previous proof.

- Now given those, get

  `t is solvable iff there exists a 01_sub gamma, (gamma t) == T0`

  Proof. For the hard direction: Suppose forall 01_sub gamma, gamma t == T0 fails. Then since every ground bterm evaluates to either T0 or T1, we have: forall 01_sub gamma, gamma t == T1. Then by the previous, t == T1. So t can't be solvable.

- Since every ground bterm evaluates to T0 or to T1, we get a very good reflection of equivalence and unifiability by reflection into bool. Note the use of = rather than == below.

  `t == T0 iff forall 01 substitutions gamma, eval_ground (gamma t) = T0`

  `t is solvable iff there exists a 01_sub gamma, eval_ground (gamma t) = T0`

```
Lemma blah :
  (T0 = T0) ∨ (T1 = T0).
Proof.
  auto.
Qed.
Lemma blah' :
  (T1 = T0) ∨ (T1 = T1).
```

```
Proof.
  auto.
Qed.
```

Main Lemma

Stated using list l, since we do induction over that

```
Lemma factor_lemma_helper :
  ∀ (l : list var) (t: bterm),
    (vars_bterm t «= l) →
    ¬ t == T0 →
    (∃ (gamma: sub),
        ls_01sub gamma ∧ (gamma ' t) == T1).
Proof.
  induction l as [| x rest IH].
  -
    intros t Vars Not0.
    assert (a:= incl_nil_eq Vars).
    assert (A1:= no_vars_Ground t a).
    ∃ const0sub.
    split; auto.
    + assert (b:= not_T0_then_T1 t A1 Not0).
      apply const0_is_01sub.
    + apply ground_T1_iff in Not0.
      – rewrite Not0.
        apply sub_T1_hom.
      – easy.
  -
    intros t Vars tNot0 .
    assert (Factor:= factor_correct t).
    unfold factor in Factor.
    assert (xNotRem := not_in_vars_remainder t x).
    assert (xNotQuot := not_in_vars_quotient t x).
    assert (HneqT0 := factor_neq_T0_strong t x tNot0).
    destruct HneqT0 as [HremNot_T0 | HquotNot_T0].
    +


      assert (VarsRem := vars_remainder_rem t x).
      assert (VarsRemIncl:= cons_rem_swap x (vars_bterm t) rest Vars).
      rewrite ← VarsRem in VarsRemIncl.
      assert (RemOK:= IH (remainder t x)
```

```
                    VarsRemIncl
                    HremNot_T0
        ).
destruct RemOK as [gammaRem [Rem_Is_01 HRemOK]].
∃ (update_sub gammaRem x T0).
assert (a:= apply_updated_sub_off
              gammaRem x T0 (remainder t x) xNotRem).
split.
×
  unfold ls_01sub.
  assert (dumb : ( (T0 = T0) ∨ (T0 = T1))). auto.
  assert (b:= update_preserves_01 gammaRem x T0 Rem_Is_01 dumb).
  easy.
×
  rewrite (Factor x).
  rewrite (sub_A_hom).
  rewrite (sub_M_hom).
  simpl.
  rewrite a.
  rewrite (apply_updated_sub_on gammaRem x T0).
  bsimp.
+
destruct HquotNot_T0 as [RemIsT0 QuotNotT0].
assert (VarsQuot := vars_quotient_rem t x).
assert (VarsQuotIncl:= cons_rem_swap x (vars_bterm t) rest Vars).
rewrite ← VarsQuot in VarsQuotIncl.
assert (QuotOK:= IH (quotient t x)
                    VarsQuotIncl
                    QuotNotT0).
destruct QuotOK as [gammaQuot HQuotOK].
destruct HQuotOK as [QuotSubs QuotEqn].
∃ (update_sub gammaQuot x T1).
assert (a:= apply_updated_sub_off
              gammaQuot x T1 (quotient t x) xNotQuot).
assert (b:= apply_updated_sub_off
              gammaQuot x T1 (remainder t x) xNotRem).
split.
×
```

```
    assert (u:= update_preserves_01 gammaQuot x T1 QuotSubs blah' ).
    easy.
  ×
    rewrite (Factor x).
    rewrite (sub_A_hom).
    rewrite (sub_M_hom).
    simpl.
    rewrite (apply_updated_sub_on).
    bsimp.
    rewrite a.
    rewrite b.
    rewrite RemIsT0.
    rewrite (sub_T0_hom).
    rewrite QuotEqn.
    now bsimp.
Qed.

Proposition factor_lemma :
  ∀ (t: bterm),
    ¬ t == T0 →
    (∃ (gamma: sub), ls_01sub gamma ∧
                              (gamma ' t) == T1).
Proof.
  intros .
  unfold ls_01sub.
  assert (a := factor_lemma_helper (vars_bterm t) t (incl_refl (vars_bterm t)) H).
  destruct a as [gamma [H1 H2]].
  destruct H1 as [l H1'].
  ∃ gamma.
  split.
  ∃ l. easy. easy.
Qed.

Proposition neqv_T0_implies :
  ∀ (t: bterm),
    ¬ t == T0 →
    (∃ (gamma: sub), ls_01sub gamma ∧
                              eval_ground (gamma ' t) = T1).
Proof.
  intros t H.
  assert (a:= factor_lemma t H).
  destruct a as [gamma [a1 a2]].
  ∃ gamma.
```

```
  split. easy.
  assert (b:= ls_01sub_gives_Ground gamma t a1).
  now apply (eval_ground_eqv_T1 (gamma ' t) b a2).
Qed.
```

This is just is just neqv_T0_implies applied to t +' T1.
Use this later in characterizing solvability.

```
Proposition neqv_T1_implies (t: bterm) :
  ¬ t == T1 →
  (∃ (gamma: sub),
      ls_01sub gamma ∧ eval_ground (gamma ' t) = T0).
Proof.
  assert (a:= neqv_T0_implies (t +' T1)).
  intros H0.
  rewrite ← T0_T1 in a.
  rewrite T0_T1 in * .
  assert (b:= a H0).
  destruct b as [gamma [H1 H2]].
  ∃ gamma.
  split. easy.
  apply eval_ground_exact_T1 in H2.
  rewrite (sub_A_hom gamma t T1) in H2.
  rewrite (sub_T1_hom gamma) in H2.
  assert (b: (gamma ' t +' T1 +' T1 == T1 +' T1)).
  rewrite H2. bsimp.
  rewrite assocA in b. rewrite invA in b. rewrite idA' in b.
  assert (c:= ls_01sub_gives_Ground gamma t H1).
  now apply (eval_ground_eqv_T0 (gamma ' t) c b).
Qed.
```

10.1.2 Characterizing equivalence in bterms of 01-subs

```
  Theorem characterizing_T0_eqv (t: bterm) :
  t == T0 ↔
  (∀ (gamma: sub),
      ls_01sub gamma → (gamma ' t) == T0).
Proof.
  split.
  - intros H gamma H1.
    now rewrite H.
  - apply classical_contra.
    intros H0.
    apply ex_not_not_all.
```

```
    assert (a:= neqv_T0_implies t H0).
    destruct a as [gamma [a1 a2]].
    ∃ gamma.
    intros H. assert (b:= H a1).
    assert (c:= eval_ground_eqv (gamma 't)).
    rewrite a2 in c.
    rewrite ← c in b.
    now apply T0_neqv_T1.
Qed.
```

Follows from the T0 result by using (t + T1) in place of t, but that proof is, sadly, more tedious than just replaying the T0 proof with obviouls variations.

```
Corollary characterizing_T1_eqv (t: bterm) :
  t == T1 ↔
  (∀ (gamma: sub),
       Is_01sub gamma → (gamma ' t) == T1).
Proof.
    split.
  - intros H gamma H1.
    now rewrite H.
  - apply classical_contra.
    intros H0.
    apply ex_not_not_all.
    assert (a:= neqv_T1_implies t H0).
    destruct a as [gamma [a1 a2]].
    ∃ gamma.
    intros H. assert (b:= H a1).
    assert (c:= eval_ground_eqv (gamma 't)).
    rewrite a2 in c.
    rewrite ← c in b.
    now apply T0_neqv_T1.
Qed.
```

10.1.3 Computability of equivalance

Improve neqv_T0_implies to get an element of the (finite) list of 01subs relevant to t

```
Theorem computable_neqv_T0_implies:
  ∀ (t : bterm),
  ¬ t == T0 →
  (∃ (gamma: sub),
       gamma el all_01subs_list (vars_bterm t) ∧
       eval_ground (gamma ' t) = T1).
Proof.
```

```
  intros t H.
  assert (a:= neqv_T0_implies t H).
  destruct a as [gamma [a1 a2]].
  assert (b:= All01_all01_bterm gamma t a1).
  destruct b as [gamma' [b1 b2]].
  ∃ gamma'.
  rewrite ← b2 in a2.
  tauto.
Qed.

Theorem computable_T0_eqv (t: bterm) :
  t == T0 ↔
  (∀ (gamma: sub),
       gamma el all_01subs_list (vars_bterm t)
       → eval_ground (gamma ' t) = T0).
Proof.
  split.
  - intros H gamma H1.
    apply (sub_compat gamma) in H.
    rewrite (sub_T0_hom gamma) in H.
    assert (a:= all_01sub_gives_Ground gamma t (vars_bterm t)H1).
    now assert (b:= eval_ground_eqv_T0 (gamma ' t) a H).
  - apply classical_contra.
    intros H0.
    assert (a:= computable_neqv_T0_implies t H0).
    destruct a as [gamma [a1 a2]].
    intros H.
    assert (b:= H gamma a1).
    assert (c:= T0_neqv_T1).
    rewrite b in a2.
    easy.
Qed.

Theorem computable_neqv_T1_implies:
  ∀ (t : bterm),
  ¬ t == T1 →
  (∃ (gamma: sub),
       gamma el all_01subs_list (vars_bterm t) ∧
       eval_ground (gamma ' t) = T0).
Proof.
  intros t H.
  assert (a:= neqv_T1_implies t H).
  destruct a as [gamma [a1 a2]].
  assert (b:= All01_all01_bterm gamma t a1).
```

```
  destruct b as [gamma' [b1 b2]].
  ∃ gamma'.
  rewrite ← b2 in a2.
  tauto.
Qed.

Theorem computable_T1_eqv (t: bterm) :
  t == T1 ↔
  (∀ (gamma: sub),
      gamma el all_01subs_list (vars_bterm t)
      → eval_ground (gamma ' t) = T1).
Proof.
  split.
  - intros H gamma H1.
    apply (sub_compat gamma) in H.
    rewrite (sub_T1_hom gamma) in H.
    assert (a:= all_01sub_gives_Ground gamma t (vars_bterm t)H1).
    now assert (b:= eval_ground_eqv_T1 (gamma ' t) a H).

  - apply classical_contra.
    intros H0.
    assert (a:= computable_neqv_T1_implies t H0).
    destruct a as [gamma [a1 a2]].
    intros H.
    assert (b:= H gamma a1).
    assert (c:= T0_neqv_T1).
    rewrite b in a2.
    easy.
Qed.

Theorem computable_eqv (t1 t2: bterm) :
  t1 == t2 ↔
  (∀ (gamma: sub),
      gamma el all_01subs_list (vars_bterm (t1 +' t2)) →
      eval_ground (gamma ' (t1 +' t2)) = T0).
Proof.
  assert (H0:= computable_T0_eqv (t1 +' t2)).
  destruct H0 as [H1 H2].
  split.
  - intros H3 gamma H4.
    rewrite eqv_eqv0 in H3.
    exact (H1 H3 gamma H4).

  - intros H.
    apply eqv_eqv0.
```

```
    exact (H2 H).
Qed.
```

Register the Proposition *t1 == t2* as an Instance of dec.

After reducing to ground-evaluations, build upon the facts that bterm-equality is decidable and that decidability is preserved by searching lists

```
Instance eqv_dec :
  ∀ (t1 t2 : bterm), dec (t1 == t2).
Proof.
  intros t1 t2.
  assert (a:= computable_eqv t1 t2).
  destruct a as [a1 a2].
  destruct (decision
              (∀ (gamma: sub),
                  gamma el all_01subs_list (vars_bterm (t1 +' t2)) →
                  eval_ground (gamma ' (t1 +' t2)) = T0)).
  - left. tauto.
  - right. tauto.
Qed.
Hint Resolve eqv_dec.
```

Boolean version

```
Definition eqvb (t1 t2: bterm) : bool :=
  dec_to_bool (decision (t1 == t2)).

Lemma eqvb_correct (t1 t2 : bterm) :
  (eqvb t1 t2 = true) ↔ t1 == t2.
Proof.
  apply dec_to_bool_correct_true.
Qed.

Lemma eqvb_reflect :
  ∀ t1 t2, reflect (t1 == t2) (eqvb t1 t2) .
Proof.
  intros t1 t2.
  apply iff_reflect.
  now rewrite eqvb_correct.
Qed.
```

10.2 Characterizing solvability

```
Lemma char_solvable_helper t :
  solvable t → ¬ t == T1.
```

```
Proof.
  intros H0 H1.
  unfold solvable in H0; unfold solves in H0; destruct H0.
  rewrite H1 in H.
  rewrite sub_T1_hom in H.
  now apply T0_neqv_T1.
Qed.

Corollary characterizing_solvability (t: bterm) :
  solvable t ↔
  (∃ (gamma: sub),
      ls_01sub gamma ∧ (gamma ' t) == T0).
Proof.
  split.
  - intros H.
    apply char_solvable_helper in H.
    assert (a:= neqv_T1_implies t H).
    firstorder.
  - intros H; destruct H as [gamma H]; destruct H.
    unfold solvable.
    now ∃ gamma.
Qed.
```

An Interesting Corollary

```
Corollary characterizing_solvability_2 (t: bterm) :
  solvable t ↔
  ¬ (t == T1).
Proof.
  split.
  - intros H0 H1.
    exact (char_solvable_helper t H0 H1).
  - intros H.
    assert (H1:= neqv_T1_implies t H).
    destruct H1 as [gamma H2].
    ∃ gamma.
    unfold solves.
    destruct H2 as [H1 H22].
    now apply eval_ground_exact_T0.
Qed.

Theorem computable_solvability (t: bterm) :
  solvable t ↔
  (∃ (gamma: sub),
      gamma el all_01subs_list (vars_bterm t) ∧
```

```
      eval_ground (gamma ' t) = T0).
Proof.
  assert (a:= characterizing_solvability t).
  destruct a as [a1 a2].
  split.
  - intros H.
    assert (b:= a1 H).
    destruct b as [gamma [b1 b2]].
    assert (c:= All01_all01_bterm gamma t b1).
    destruct c as [gamma' [c1 c2]].
    ∃ gamma'.
    split. easy.
    rewrite ← c2 in b2.
    assert (d:= all_01sub_gives_Ground gamma' t (vars_bterm t) c1).
    now apply (eval_ground_eqv_T0 (gamma' ' t) d b2).
  - intros H.
    apply a2.
    destruct H as [gamma [H1 H2]].
    assert (b:= all_01sub_gives_Ground gamma t (vars_bterm t) H1).
    ∃ gamma.
    split.
    + unfold ls_01sub.
      ∃ (vars_bterm t).
      now apply (all01_All01 (vars_bterm t) gamma H1).
    + now apply (eval_ground_exact_T0).
Defined.

Instance solvable_dec :
  ∀ (t : bterm), dec (solvable t).
Proof.
  intros t.
  assert (a:= computable_solvability t).
  destruct a as [a1 a2].
  destruct (decision
              (∃ gamma : sub,
                  gamma el all_01subs_list (vars_bterm t) ∧
                  eval_ground (gamma ' t) = T0)).
  - left. tauto.
  - right. tauto.
Defined.
Hint Resolve solvable_dec.

Definition solvable_bool (t : bterm) : bool :=
  dec_to_bool (decision (solvable t)).
```

```
Lemma solvable_bool_correct (t : bterm) :
  (solvable_bool t = true) ↔ solvable t.
Proof.
  apply dec_to_bool_correct_true.
Qed.
```

Chapter 11

Library BooleanRings.Lowenheim

```
Require Export Bool.Sumbool.
Require Export Omega.
Require Export List.
Export ListNotations.
Require Export Basics.
Export Setoid.
Require Export Logic.FunctionalExtensionality.
Require Export Utilities.
Require Export TheoryDef.
Require Export BTermsUtilities.
Require Export Substitutions.
Require Export Factoring.
Require Export EquivalenceAndSolvability.
```

11.1 The Algorithm

Compute an initial solution

Returns (Some gamma) if gamma is a 01_sub making t eval to T0

```
Definition ground_soln (t: bterm) : option sub :=
  find (fun s ⇒ eqvb (s ' t) T0) (
        all_01subs_bterm t).
```

Build a new sub out of a given sub and a given bterm

```
Definition lowenheim_lift (t: bterm) (tau: sub) (x: var) : bterm :=
  if inb x (vars_bterm t)
  then ((t +' T1) *' (V x) +' t *' (tau ' (V x)) )
  else (V x).
```

```
Hint Unfold lowenheim_lift.
```

Amazingly, this constructs an mgu

```
Definition solve_lowenheim (t: bterm) :=
  option_map (lowenheim_lift t) (ground_soln t).
```

11.2 Correctness Argument

11.2.1 Facts About lowenheim_lift

```
Lemma lowenheim_lift_var_In :
  ∀ (t: bterm) (tau: sub) (x: var),
    In x (vars_bterm t) →
    (lowenheim_lift t tau) ' (V x) ==
    (t +' T1) *' (V x) +' t *' (tau ' (V x)).
Proof.
  intros t tau x H.
  unfold lowenheim_lift.
  simpl.
  destruct (natlist_reflect x (vars_bterm t)).
  easy. now exfalso.
Qed.

Lemma lowenheim_lift_bterm:
  ∀ (t: bterm) (tau : sub),
  ∀ (u: bterm),
    (vars_bterm u) «= (vars_bterm t) →
    (lowenheim_lift t tau ) ' u ==
    (t+'T1) *' u +' t *' (tau ' u).
Proof.
  intros t tau u Hvars.
  induction u as [ | | x | t1 IH1 t2 IH2| t1 IH1 t2 IH2]; simpl; auto.
  unfold lowenheim_lift.
  - repeat rewrite sub_T0_hom; auto.
    repeat rewrite zeroM'. now rewrite invA.
  - repeat rewrite sub_T1_hom; auto.
    repeat rewrite idM'. rewrite comA. rewrite ← assocA.
    now rewrite invA.
  - unfold lowenheim_lift.
    decide (In x (vars_bterm t)).
    + now apply lowenheim_lift_var_In.
    + simpl.
```

```
      destruct (natlist_reflect x (vars_bterm t)).
      easy.
      assert (a: x el vars_bterm t).
      – simpl in Hvars.
        apply (Hvars x (in_sing_refl x)).
      – tauto.
  - intros. simpl. bsimp.
    rewrite IH1. rewrite IH2.
    bsimp.
    assert (a:= (vars_bterm_incl_A_R t1 t2)).
    now apply incl_tran with (vars_bterm (t1 +' t2)).
    assert (a:= (vars_bterm_incl_A_L t1 t2)).
    now apply incl_tran with (vars_bterm (t1 +' t2)).
  - intros. simpl.
    rewrite IH1. rewrite IH2.
    intros.
    bsimp.
    assert (a:= (vars_bterm_incl_M_R t1 t2)).
    now apply incl_tran with (vars_bterm (t1 +' t2)).
    assert (a:= (vars_bterm_incl_M_L t1 t2)).
    now apply incl_tran with (vars_bterm (t1 +' t2)).
Qed.

Theorem lift_unifies : ∀ (t: bterm) (tau: sub),
    solves tau t →
    solves (lowenheim_lift t tau) t.
Proof.
  intros t tau H.
  assert (a:= lowenheim_lift_bterm t tau t (incl_refl (vars_bterm t))).
  unfold solves in * .
  rewrite a.
  rewrite H.
  bsimp.
Qed.

Lemma lift_mgu : ∀ (t: bterm) (tau: sub),
    solves tau t →
    mgu_strong (lowenheim_lift t tau) t.
Proof.
  intros t tau Htau.
  unfold mgu.
  split.
  -
    now apply lift_unifies.
```

```
  -
    intros gamma Hgamma .
    unfold sub_leq_strong.
    unfold compose_sub.
    unfold lowenheim_lift.
    intros x.
    destruct (natlist_reflect x (vars_bterm t)).
    unfold solves in Hgamma.
    rewrite (sub_A_hom).
    rewrite (sub_M_hom).
    rewrite (sub_A_hom).
    bsimp.
    rewrite (sub_T1_hom).
    rewrite idM'.
    rewrite Hgamma.
    rewrite zeroM.
    rewrite (sub_M_hom).
    rewrite Hgamma.
    rewrite zeroM.
    now bsimp. easy.
Qed.
```

11.2.2 Solvable iff There is a Ground Solution

```
Section Solvable_Ground_Soln.
  Variable t: bterm.
  Lemma solvable_then_ground_soln :
    solvable t →
    ∃ gamma, ground_soln t = Some gamma.
  Proof.
    intros H.
    apply existsb_find.
    apply List.existsb_exists.
    rewrite (computable_solvability t) in H.
    destruct H as [gamma H1]. destruct H1.
    assert (H1 := eval_ground_exact_T0 (gamma ' t) H0).
    ∃ gamma.
    split.
    - easy.
    - now rewrite eqvb_correct.
  Qed.
```

```
  Lemma ground_soln_then_solvable :
    (∃ gamma, ground_soln t = Some gamma) →
    solvable t .
  Proof.
    intros H. destruct H as [gamma H1].
    ∃ gamma.
    unfold ground_soln in H1.
    assert (H2 := find_some _ _ H1).
    destruct H2.
    rewrite (eqvb_correct (gamma ' t) T0) in H0.
    easy.
  Qed.

  Lemma solvable_iff_ground_soln :
    solvable t ↔
    ∃ gamma, ground_soln t = Some gamma.
  Proof.
    split.
    apply solvable_then_ground_soln.
    apply ground_soln_then_solvable.
  Qed.

End Solvable_Ground_Soln.
```

11.2.3 Correctness of the None case

```
Lemma lowenheim_correct_None t :
  solve_lowenheim t = None →
  solvable t → False.
Proof.
  intros H1 H2.
  rewrite (solvable_iff_ground_soln t) in H2.
  apply option_map_reflect_None in H1.
  assert (A1:= None_not_Some ground_soln t).
  assert (A2:= A1 H1).
  destruct H2 as [gamma H2'].
  congruence.
Qed.
```

11.2.4 Correctness of the Some case

```
Lemma ground_soln_Some t gamma :
  ground_soln t = Some gamma →
  solves gamma t.
```

```
Proof.
  intros H.
  unfold ground_soln in H.
  assert (H1:= find_some (fun s ⇒ eqvb (s ' t) T0)
                         (all_01subs_bterm t) H ).
  destruct H1.
  now rewrite eqvb_correct in H1.
Qed.

Lemma solve_lowenheim_Some (t: bterm) (sigma: sub) :
  (solve_lowenheim t = Some sigma) →
  ∃ gamma,
    ground_soln t = Some gamma ∧
    sigma = (lowenheim_lift t) gamma.
Proof.
  intros.
  unfold solve_lowenheim in H.
  assert (A1:= option_map_reflect_Some
                 (lowenheim_lift t)
                 (ground_soln t) sigma H).
  destruct A1 as [gamma A11]. destruct A11 as [A111 A112].
  now ∃ gamma.
Qed.

Lemma lowenheim_correct_Some sigma t :
  solve_lowenheim t = Some sigma →
  mgu_strong sigma t.
Proof.
  intro H.
  assert (a:= solve_lowenheim_Some t sigma H).
  elim a.
  intros.
  elim H0.
  intros.
  rewrite H2.
  apply lift_mgu.
  now apply (ground_soln_Some t).
Qed.
```

11.2.5 Put the pieces together

```
Theorem lowenheim_correct (t: bterm) :
  match (solve_lowenheim t) with
  | None ⇒ ¬ (solvable t)
```

```
  | Some sigma ⇒ mgu_strong sigma t
  end .
Proof.
  case_eq (solve_lowenheim t).
  - intros sigma.
    exact (lowenheim_correct_Some sigma t).
  - exact (lowenheim_correct_None t).
Qed.
```

Chapter 12

Library BooleanRings.VarElim

```
Require Export Omega.
Require Export List.
Export ListNotations.
Export Setoid.

Require Export Utilities.
Require Export TheoryDef.
Require Export BTermsUtilities.
Require Export Substitutions.
Require Export Factoring.
Require Export EquivalenceAndSolvability.
```

12.1 The Variable Elimination algorithm

We build on the Factoring module, where, for any bterm t and variable x we define a quotient q and remainder r, such that t == qx + r. Then the variable elimination algorithm for solvability is

- choose a variable x from vars of t, and write t as qx + r
- set t' to be (q+1) r
- compute a reproductive sigma' solving t'
- then the following sigma is a reproductive unifier of t

  `sigma' + {x |-> x * (sigma' q) + x + (sigma r) | x in vars t}`

12.2 Computing a unifier

The bterm for the recursive call: (q+1) * r

```
Definition derived_bterm (t: bterm) (x: var) : bterm :=
  ((quotient t x) +' T1 ) *' (remainder t x).
```

The binding for x after the recursive call

```
Definition var_image (sigma: sub) (x: var) (q r: bterm) : bterm :=
  ( (V x) *' ( (sigma ' q) +' T1) ) +' (sigma ' r).
```

The updated substitution from the answer to the recursive call

```
Definition var_elim_update (sigma: sub) (x: var) (q r: bterm) : sub :=
  (update_sub sigma x (var_image sigma x q r)).
```

12.2.1 Main Function

Familiar pattern: helper defined for an arbitrary list of vars, real function specializies to vars in the bterms

```
Fixpoint varElimX (t: bterm) (vars: list var) : option sub :=
  match vars with
  | [] ⇒ if eqvb t T0 then Some id_sub else None
  | (x::xs) ⇒
    match varElimX (((quotient t x) +' T1) *' (remainder t x) ) xs with
    | None ⇒ None
    | Some sigma ⇒
      Some (var_elim_update sigma x (quotient t x) (remainder t x))
    end
  end.
```

for now don't bother with simplify_sub

```
Definition varElim (t: bterm) : option sub :=
  varElimX t (vars_bterm t).
```

12.3 Correctness Argument

Proof outline:

- As we go from t down through the recursive calls, if t is solvable then each derived bterm is solvable. (Lemma solvability_preserved)
- The base case, of a closed bterm, is either unsolvable or has the identity as an mgu
- If sigma is an mgu the derived bterm, then the updated sub is a reproductive solution to the original bterm Lemma ??
- So if t is solvable at all, it has an mgu, found by the algorithm

12.3.1 Behavior of the updated sub

Action of the updated sub on the chosen var x

```
Lemma update_on_var sigma x q r :
  (var_elim_update sigma x q r) x ==
  (var_image sigma x q r).
Proof.
  unfold var_elim_update.
  now rewrite apply_updated_sub_on.
Qed.
```

Updated sub agrees with the old sub on the quotient and remainder

```
Lemma update_on_quotient : ∀ x sigma t,
    (var_elim_update sigma x (quotient t x) (remainder t x)) ' (quotient t x) ==
    sigma ' (quotient t x).
Proof.
  intros x sigma t.
  unfold var_elim_update.
  now rewrite (apply_updated_sub_off sigma x _ (quotient t x) (not_in_vars_quotient t x)).
Qed.

Lemma update_on_remainder : ∀ x sigma t,
    (var_elim_update sigma x (quotient t x) (remainder t x)) ' (remainder t x) ==
    sigma ' (remainder t x).
Proof.
  intros x sigma t.
  unfold var_elim_update.
  now rewrite (apply_updated_sub_off sigma x _ (remainder t x) (not_in_vars_remainder t
x)).
Qed.
```

12.3.2 Variables in the derived bterm

```
Lemma derived_bterm_vars (x: var) (xs: list var) (t: bterm) :
  vars_bterm t «= x :: xs →
  vars_bterm (derived_bterm t x) «= xs .
Proof.
  intros H.
  unfold derived_bterm.
  simpl; rewrite app_nil_r.
  apply incl_app.
  - assert (A:= cons_rem_swap x (vars_bterm t) xs H).
    assert (B:= vars_quotient_rem t x).
    now apply incl_tran with (rem (vars_bterm t) x).
  - assert (A:= cons_rem_swap x (vars_bterm t) xs H).
    assert (B:= vars_remainder_rem t x).
    now apply incl_tran with (rem (vars_bterm t) x).
Qed.
```

12.3.3 Solvability preserved in derived bterm

Helper for *solvability-preserved*

```
Lemma solvability_preserved_helper: ∀ q x r ,
    x *' q +' r == T0 →
    (q +' T1) *' r == T0.
Proof.
  intros q x r H.
  cut ((q +' T1) *' (x *' q +' r) == (q +' T1) *' T0).
  intros.
  bsimp.
  - ring_simplify in H0.
    rewrite ord2M in H0.
    rewrite invA in H0.
    now rewrite idA in H0.
  - rewrite H.
    bsimp.
Qed.

Proposition solvability_preserved (t: bterm) (x: var) (tau: sub) :
  solves tau t →
  solves tau (derived_bterm t x).
Proof.
  intros Hsolves.
  unfold solves in *.
```

```
  rewrite (factor_correct t x) in Hsolves.
  unfold factor in Hsolves.

  rewrite sub_A_hom in Hsolves.
  rewrite sub_M_hom in Hsolves.

  unfold derived_bterm.
  rewrite sub_M_hom.
  now apply solvability_preserved_helper in Hsolves.
Qed.

Corollary unsolvability_reflected (t: bterm) (x: var) :
  ¬ solvable (derived_bterm t x) → ¬ solvable t.
Proof.
  intros H1 H2.
  unfold solvable in *.
  destruct H2 as [tau H21].
  assert (A:= solvability_preserved t x tau H21).
  firstorder.
Qed.
```

NOTE! sigma' is the unifier at the recursive call, sigma is the updated one
Proof:

- (sigma t) = (sigma (x q + r) =
  (x * ((sigma' q) + 1) + (sigma' r) * (sigma' r) * (sigma' s))
- multiply that out, eventually get to

(sigma (q+1)* r) = (sigma t') = 0

12.3.4 Solvability reflected by derived bterm

```
Lemma new_unifier_works : ∀ (t: bterm) (x: var) (delta: sub),
     solves delta (derived_bterm t x) →
     solves (var_elim_update delta x (quotient t x) (remainder t x )) t.
Proof.
  intros t x delta Hsolves.

  remember (quotient t x) as q.
  remember (remainder t x) as r.
  remember (var_elim_update delta x (quotient t x) (remainder t x)) as sigma'.
```

suffices to show updated solves (factor t)

```
  apply (solves_compat (var_elim_update delta x q r) (factor t x)).

  - symmetry; apply factor_correct.
```

```
  - unfold factor. unfold solves.
    rewrite ← Heqq; rewrite ← Heqr.
    rewrite sub_A_hom; rewrite sub_M_hom.
    assert (a:= (update_on_var delta x q r )).
    rewrite a.
    unfold var_image.
    assert (b:= (update_on_quotient x delta t )).
    rewrite Heqq. rewrite Heqr.
    rewrite b.
    assert (c:= (update_on_remainder x delta t)).
    rewrite c.
    bsimp; rewrite assocM; rewrite ord2M; rewrite invA; bsimp.
    assert (tmp: ∀ q r, (q *' r +' r) == (q +' T1) *' r).
    intros; bsimp.
    rewrite tmp.
    unfold derived_bterm in Hsolves.
    rewrite ← Heqq in Hsolves.
    rewrite ← Heqr in Hsolves.
    rewrite ← Heqq. rewrite ← Heqr.
    apply Hsolves.
Qed.
```

Proof:

- let tau solve t
- assuming sigma' mgu , show that for all vars y,

(tau (sigma y)) = tau y

- since sigma' mgu, have, for all y,

(tau (sigma' y)) = tau y

- in case: y is not x: sigma agrees with sigma'. OK
- in case y is x:

(tau (sigma x)) = ...multiply it out
The update of sigma agrees with sigma away from x

```
Lemma sigma_sigma'_agree (sigma: sub) x q r y:
  ¬ (x = y) →
  sigma ' (V y) = (var_elim_update sigma x q r) ' (V y).
```

```
Proof.
  intros Hneq.
  unfold var_elim_update.
  symmetry.
  apply (apply_updated_sub_off_var sigma x y _ Hneq).
Qed.
```

Helper: the interesting case for showing mgu

Another example of something that is just routine equational reasoning, but is manifest here as tedious rewriting. Need a boolean-ring version of the ring tactic!!

```
Lemma new_unifier_mgu_help t tau sigma' x :
  mgu_strong sigma' (derived_bterm t x) →
  solves tau t →
  tau ' (var_elim_update sigma' x (quotient t x) (remainder t x) x)
  == tau x.
Proof.
  remember (quotient t x) as q eqn:qeq.
  remember (remainder t x) as r eqn:req.

  intros Hrepu Hsolves_tau_t.
  assert (p:= solvability_preserved t x tau Hsolves_tau_t).
  unfold mgu_strong in Hrepu.
  destruct Hrepu as [H1 H2].

  unfold var_elim_update.
  unfold update_sub.
  rewrite ← beq_nat_refl.
  unfold var_image.

  rewrite sub_A_hom.
  rewrite sub_M_hom.
  rewrite sub_A_hom.
  rewrite sub_T1_hom.
  bsimp.

  assert (a:= H2 tau p).
  unfold sub_leq_strong in a.
  unfold compose_sub in a.

  assert (b:= compose_bterms sigma' tau tau ).
  unfold compose_sub in b.
  assert (bq := b a q).
  assert (br := b a r).
  rewrite bq.
  rewrite br.

  unfold solves in p.
```

```
  unfold derived_bterm in p.
  rewrite ← qeq in p. rewrite ← req in p.
  rewrite (sub_M_hom tau (q +' T1) r) in p.
  rewrite (sub_A_hom) in p.
  rewrite (sub_T1_hom) in p.

  rewrite assocA.
  rewrite comA.
  rewrite assocA.
  assert (c: (tau ' r +' (tau ' (V x)) *' (tau ' q) == T0)).

  rewrite ← sub_M_hom.
  rewrite ← sub_A_hom.
  assert (d: (r +' V x *' q == t)).
  assert (e:= factor_correct t x).
  unfold factor in e.
  rewrite ← qeq in e. rewrite ← req in e.
  rewrite e.
  bsimp.
  rewrite d; easy.
  rewrite c; easy.
  Qed.

Proposition new_unifier_mgu : ∀ (t: bterm) (x: var) (sigma: sub),
     mgu_strong sigma (derived_bterm t x) →
     mgu_strong (var_elim_update sigma x (quotient t x) (remainder t x )) t.
Proof.
  intros t x sigma H_mgu_sigma.
  remember H_mgu_sigma as H_mgu_sigma_rem.

  unfold mgu_strong in H_mgu_sigma.
  destruct H_mgu_sigma as [H1 H2].

  remember (quotient t x) as q eqn:qeq.
  remember (remainder t x) as r eqn:req.
  split.
  - subst. now apply new_unifier_works.
  - intros tau H_tau_solves_t.

   since tau solves t, it solves t'

    assert (H_tau_solves_t' :=
                solvability_preserved t x tau H_tau_solves_t).

    assert (a:= H2 tau H_tau_solves_t').
    unfold sub_leq_strong.

    intros y.
    decide (y = x).
```

```
    + Case x=y          rewrite e.
      unfold compose_sub.
      assert (b:= new_unifier_mgu_help t tau sigma).
      unfold mgu.
      assert (c:= b x H_mgu_sigma_rem H_tau_solves_t).
      now subst.
    + Case x <> y          unfold compose_sub.
      unfold var_elim_update.
      apply neq_comm in n.
      assert (b:= sigma_sigma'_agree sigma x q r y n).
      unfold update_sub.
      rewrite beq_nat_false'.
      unfold sub_leq_strong in a.
      unfold compose_sub in a.
      now apply (a y). easy.
Qed.
```

12.3.5 Correctness of the None case

```
Lemma varElimX_None_backwards (x: var) (xs: list var) (t: bterm) :
  varElimX t (x::xs) = None →
  varElimX (derived_bterm t x) xs = None.
Proof.
  intros H.
  inversion H.
  unfold derived_bterm.
  destruct ( varElimX ((quotient t x +' T1) *' remainder t x) xs ).
  inversion H1.
  easy.
Qed.

Lemma varElimX_correct_None (vars: list var) :
  ∀ (t: bterm),
    varElimX t vars = None →
    (vars_bterm t) «= vars →
    ¬ solvable t.
Proof.
  induction vars as [| v vars' IH].
  - intros t H1 H2 H3.
    apply incl_nil_eq in H2.
    assert (A:= no_vars_Ground t H2).
    simpl in H1.
    destruct (eqvb_reflect t T0).
```

```
    + inversion H1.
    + apply characterizing_solvability_2 in H3.
      assert (B:= Ground_T0_or_T1 t A).
      now inversion B.
  - intros t H1 H2.
    apply varElimX_None_backwards in H1.
    assert (A:= derived_bterm_vars v vars' t H2).
    assert (B:= IH (derived_bterm t v) H1 A).
    now apply (unsolvability_reflected t v).
Qed.

Proposition varElim_correct_None t :
  varElim t = None →
  ¬ solvable t.
Proof.
  intros H.
  unfold varElim in H.
  now apply varElimX_correct_None with (vars_bterm t).
Qed.
```

12.3.6 Correctness of the Some case

```
Lemma varElimX_Some_backwards (x: var) (xs: list var) (t: bterm) sigma :
  varElimX t (x::xs) = Some sigma →
  (∃ sigma', varElimX (derived_bterm t x) xs = Some sigma'
                    ∧ sigma = (var_elim_update sigma'
                                               x
                                               (quotient t x)
                                               (remainder t x))).
Proof.
  intros H.
  inversion H.
  unfold derived_bterm.
  destruct ( varElimX ((quotient t x +' T1) *' remainder t x) xs ).
  inversion H1.
  - ∃ s; easy.
  - easy.
Qed.

Lemma varElimX_correct_Some (vars: list var) :
  ∀ (t: bterm) (upd: sub),
    varElimX t vars = Some upd →
    (vars_bterm t) ⊆= vars →
    mgu_strong upd t.
```

```
Proof.
  induction vars as [| x xs IH].
  - intros t upd H1 H2.
    apply incl_nil_eq in H2.
    assert (A:= no_vars_Ground t H2).
    simpl in H1.
    destruct (eqvb_reflect t T0).
    + inversion H1.
      now apply mgu_T0.
    + easy.
  - intros t upd H1 H2.
    apply varElimX_Some_backwards in H1.
    destruct H1 as [sigma' H11].
    destruct H11 as [l r].
    assert (A:= derived_bterm_vars x xs t H2).
    assert (B:= IH (derived_bterm t x) sigma' l A ).
    rewrite r.
    now apply new_unifier_mgu in B.
Qed.

Proposition varElim_correct_Some t sigma :
  varElim t = Some sigma →
  mgu_strong sigma t.
Proof.
  intros H.
  unfold varElim in H.
  now apply varElimX_correct_Some with (vars_bterm t).
Qed.
```

12.3.7 Put the pieces together

```
Theorem varElim_correctness (t: bterm) :
  match (varElim t) with
  | None ⇒ ¬ solvable t
  | Some sigma ⇒ mgu_strong sigma t
  end.
Proof.
  case_eq (varElim t).
  - intros sigma.
    exact (varElim_correct_Some t sigma ).
  - exact (varElim_correct_None t).
Qed.
```

Chapter 13

Library BooleanRings.Smolka_Base

13.1 Base Library for ICL

- Version: 27 April 2015
- Author: Gert Smolka, Saarland University
- Acknowlegments: Sigurd Schneider, Dominik Kirst

```
Global Set Implicit Arguments.
Global Unset Strict Implicit.
```

```
Require Export Omega List Morphisms.
```

```
Ltac inv H := inversion H; try subst; clear H.
```

13.2 De Morgan laws

```
Lemma DM_or (X Y : Prop) :
  ¬ (X ∨ Y) ↔ ¬ X ∧ ¬ Y.
Proof.
  tauto.
Qed.
```

```
Lemma DM_exists X (p : X → Prop) :
  ¬ (∃ x, p x) ↔ ∀ x, ¬ p x.
Proof.
  firstorder.
Qed.
```

13.3 Size recursion

```
Lemma size_recursion (X : Type) (sigma : X → nat) (p : X → Type) :
  (∀ x, (∀ y, sigma y < sigma x → p y) → p x) →
  ∀ x, p x.
Proof.
  intros D x. apply D.
  cut (∀ n y, sigma y < n → p y).
  now eauto.
  clear x. intros n.
  induction n; intros y E.
  - exfalso; omega.
  - apply D. intros x F. apply IHn. omega.
Qed.
```

Arguments size_recursion {X} sigma {p} _ _.

13.4 Iteration

```
Section Iteration.
  Variable X : Type.
  Variable f : X → X.

  Fixpoint it (n : nat) (x : X) : X :=
    match n with
      | 0 ⇒ x
      | S n' ⇒ f (it n' x)
    end.

  Lemma it_ind (p : X → Prop) x n :
    p x → (∀ z, p z → p (f z)) → p (it n x).
  Proof.
    intros A B. induction n; simpl; auto.
  Qed.

  Definition FP (x : X) : Prop := f x = x.

  Lemma it_fp (sigma : X → nat) x :
    (∀ n, FP (it n x) ∨ sigma (it n x) > sigma (it (S n) x)) →
    FP (it (sigma x) x).
  Proof.
    intros A.
    assert (B: ∀ n, FP (it n x) ∨ sigma x ≥ n + sigma (it n x)).
    { intros n; induction n; simpl.
      - auto.
```

```
    - destruct IHn as [B|B].
      + left. now rewrite B.
      + destruct (A n) as [C|C].
        × left. now rewrite C.
        × right. simpl in C. omega. }
  destruct (A (sigma x)), (B (sigma x)); auto; exfalso; omega.
Qed.
End Iteration.
```

13.5 Decidability

```
Definition dec (X : Prop) : Type := {X} + {¬ X}.
Notation "'eq_dec' X" := (∀ x y : X, dec (x=y)) (at level 70).

Existing Class dec.
Definition decision (X : Prop) (D : dec X) : dec X := D.
Arguments decision X {D}.
Tactic Notation "decide" constr(p) :=
  destruct (decision p).
Tactic Notation "decide" constr(p) "as" simple_intropattern(i) :=
  destruct (decision p) as i.

Hint Extern 4 ⇒
match goal with
  | [ ⊢ dec ?p ] ⇒ exact (decision p)
end.

Hint Extern 4 ⇒
match goal with
  | [ ⊢ dec ((fun _ ⇒ _) _) ] ⇒ simpl
end : typeclass_instances.

Instance True_dec : dec True :=
  left I.

Instance False_dec : dec False :=
  right (fun A ⇒ A).

Instance impl_dec (X Y : Prop) :
  dec X → dec Y → dec (X → Y).
Proof.
  unfold dec; tauto.
Defined.

Instance and_dec (X Y : Prop) :
```

```
  dec X → dec Y → dec (X ∧ Y).
Proof.
  unfold dec; tauto.
Defined.

Instance or_dec (X Y : Prop) :
  dec X → dec Y → dec (X ∨ Y).
Proof.
  unfold dec; tauto.
Defined.

Instance not_dec (X : Prop) :
  dec X → dec (¬ X).
Proof.
  unfold not. auto.
Defined.

Instance iff_dec (X Y : Prop) :
  dec X → dec Y → dec (X ↔ Y).
Proof.
  unfold iff. auto.
Qed.

Lemma dec_DN X :
  dec X → ¬¬ X → X.
Proof.
  unfold dec; tauto.
Qed.

Lemma dec_DM_and X Y :
  dec X → dec Y → ¬ (X ∧ Y) → ¬ X ∨ ¬ Y.
Proof.
  unfold dec; tauto.
Qed.

Lemma dec_DM_impl X Y :
  dec X → dec Y → ¬ (X → Y) → X ∧ ¬ Y.
Proof.
  unfold dec; tauto.
Qed.

Lemma dec_prop_iff (X Y : Prop) :
  (X ↔ Y) → dec X → dec Y.
Proof.
  unfold dec; tauto.
Defined.

Instance bool_eq_dec :
```

```
  eq_dec bool.
Proof.
  intros x y. hnf. decide equality.
Defined.

Instance nat_eq_dec :
  eq_dec nat.
Proof.
  intros x y. hnf. decide equality.
Defined.

Instance nat_le_dec (x y : nat) : dec (x ≤ y) :=
  le_dec x y.
```

13.6 Lists

```
Definition equi X (A B : list X) : Prop :=
  incl A B ∧ incl B A.

Hint Unfold equi.

Export ListNotations.
Notation "| A |" := (length A) (at level 65).
Notation "x 'el' A" := (In x A) (at level 70).
Notation "A «= B" := (incl A B) (at level 70).
Notation "A === B" := (equi A B) (at level 70).
```

Register additional simplification rules with autorewrite / simpl_list

```
Hint Rewrite ← app_assoc : list.
Hint Rewrite rev_app_distr map_app prod_length : list.

Lemma list_cycle (X : Type) (A : list X) x :
  x::A ≠ A.
Proof.
  intros B.
  assert (C: |x::A| ≠ |A|) by (simpl; omega).
  apply C. now rewrite B.
Qed.
```

13.7 Decidability laws for lists

```
Instance list_eq_dec X :
  eq_dec X → eq_dec (list X).
Proof.
```

```
  intros D. apply list_eq_dec. exact D.
Defined.

Instance list_in_dec (X : Type) (x : X) (A : list X) :
  eq_dec X → dec (x el A).
Proof.
  intros D. apply in_dec. exact D.
Defined.

Lemma list_sigma_forall X A (p : X → Prop) (p_dec : ∀ x, dec (p x)) :
  {x | x el A ∧ p x} + {∀ x, x el A → ¬ p x}.
Proof.
  induction A as [|x A]; simpl.
  - tauto.
  - destruct IHA as [[y [D E]]|D].
    + eauto.
    + destruct (p_dec x) as [E|E].
      × eauto.
      × right. intros y [[]|F]; auto.
Defined.

Arguments list_sigma_forall {X} A p {p_dec}.

Instance list_forall_dec X A (p : X → Prop) :
  (∀ x, dec (p x)) → dec (∀ x, x el A → p x).
Proof.
  intros p_dec.
  destruct (list_sigma_forall A (fun x ⇒ ¬ p x)) as [[x [D E]]|D].
  - right. auto.
  - left. intros x E. apply dec_DN; auto.
Defined.

Instance list_exists_dec X A (p : X → Prop) :
  (∀ x, dec (p x)) → dec (∃ x, x el A ∧ p x).
Proof.
  intros p_dec.
  destruct (list_sigma_forall A p) as [[x [D E]]|D].
  - unfold dec. eauto.
  - right. intros [x [E F]]. exact (D x E F).
Defined.

Lemma list_exists_DM X A (p : X → Prop) :
  (∀ x, dec (p x)) →
  ¬ (∀ x, x el A → ¬ p x) → ∃ x, x el A ∧ p x.
Proof.
  intros D E.
  destruct (list_sigma_forall A p) as [F|F].
```

```
  + destruct F as [x F]. eauto.
  + contradiction (E F).
Qed.

Lemma list_exists_not_incl X (A B : list X) :
  eq_dec X →
  ¬ A «= B → ∃ x, x el A ∧ ¬ x el B.
Proof.
  intros D E.
  apply list_exists_DM; auto.
  intros F. apply E. intros x G.
  apply dec_DN; auto.
Qed.

Lemma list_cc X (p : X → Prop) A :
  (∀ x, dec (p x)) →
  (∃ x, x el A ∧ p x) → {x | x el A ∧ p x}.
Proof.
  intros D E.
  destruct (list_sigma_forall A p) as [[x [F G]]|F].
  - eauto.
  - exfalso. destruct E as [x [G H]]. apply (F x); auto.
Defined.
```

13.8 Membership

We use the following lemmas from Coq's standard library List.

- *in_eq* : x *el* x::A
- *in_nil* : ¬ x *el* nil
- in_cons : x *el* A → x *el* y::A
- in_or_app : x *el* A ∨ x *el* B → x *el* A++B
- in_app_iff : x *el* A++B ↔ x *el* A ∨ x *el* B
- in_map_iff : y *el* map f A ↔ ∃ x, f $x = y$ ∧ x *el* A

```
Hint Resolve in_eq in_nil in_cons in_or_app.

Section Membership.
  Variable X : Type.
  Implicit Types x y : X.
  Implicit Types A B : list X.
```

```
Lemma in_sing x y :
  x el [y] → x = y.
Proof.
  simpl. intros [[]|[]]. reflexivity.
Qed.

Lemma in_cons_neq x y A :
  x el y::A → x ≠ y → x el A.
Proof.
  simpl. intros [[]|D] E; congruence.
Qed.

Lemma not_in_cons x y A :
  ¬ x el y :: A → x ≠ y ∧ ¬ x el A.
Proof.
  intuition; subst; auto.
Qed.
```

13.9 Disjointness

```
Definition disjoint A B :=
  ¬ ∃ x, x el A ∧ x el B.

Lemma disjoint_forall A B :
  disjoint A B ↔ ∀ x, x el A → ¬ x el B.
Proof.
  split.
  - intros D x E F. apply D. ∃ x. auto.
  - intros D [x [E F]]. exact (D x E F).
Qed.

Lemma disjoint_symm A B :
  disjoint A B → disjoint B A.
Proof.
  firstorder.
Qed.

Lemma disjoint_incl A B B' :
  B' «= B → disjoint A B → disjoint A B'.
Proof.
  firstorder.
Qed.

Lemma disjoint_nil B :
  disjoint nil B.
Proof.
```

```
  firstorder.
Qed.

Lemma disjoint_nil' A :
  disjoint A nil.
Proof.
  firstorder.
Qed.

Lemma disjoint_cons x A B :
  disjoint (x::A) B ↔ ¬ x el B ∧ disjoint A B.
Proof.
  split.
  - intros D. split.
    + intros E. apply D. eauto.
    + intros [y [E F]]. apply D. eauto.
  - intros [D E] [y [[F|F] G]].
    + congruence.
    + apply E. eauto.
Qed.

Lemma disjoint_app A B C :
  disjoint (A ++ B) C ↔ disjoint A C ∧ disjoint B C.
Proof.
  split.
  - intros D. split.
    + intros [x [E F]]. eauto 6.
    + intros [x [E F]]. eauto 6.
  - intros [D E] [x [F G]].
    apply in_app_iff in F as [F|F]; eauto.
Qed.

End Membership.

Hint Resolve disjoint_nil disjoint_nil'.
```

13.10 Inclusion

We use the following lemmas from Coq's standard library List.

- incl_refl : A «= A
- *incl_tl* : A «= B → A «= x::B
- incl_cons : x *el* B → A «= B → x::A «= B
- *incl_appl* : A «= B → A «= B++C

- *incl_appr* : A «= C → A «= B++C
- incl_app : A «= C → B «= C → A++B «= C

```
Hint Resolve incl_refl incl_tl incl_cons incl_appl incl_appr incl_app.

Lemma incl_nil X (A : list X) :
  nil «= A.

Proof. intros x []. Qed.

Hint Resolve incl_nil.

Lemma incl_map X Y A B (f : X → Y) :
  A «= B → map f A «= map f B.

Proof.
  intros D y E. apply in_map_iff in E as [x [E E']].
  subst y. apply in_map_iff. eauto.
Qed.

Section Inclusion.
  Variable X : Type.
  Implicit Types A B : list X.

  Lemma incl_nil_eq A :
    A «= nil → A=nil.

  Proof.
    intros D. destruct A as [|x A].
    - reflexivity.
    - exfalso. apply (D x). auto.
  Qed.

  Lemma incl_shift x A B :
    A «= B → x::A «= x::B.

  Proof. auto. Qed.

  Lemma incl_lcons x A B :
    x::A «= B ↔ x el B ∧ A «= B.
  Proof.
    split.
    - intros D. split; hnf; auto.
    - intros [D E] z [F|F]; subst; auto.
  Qed.

  Lemma incl_sing x A y :
    x::A «= [y] → x = y ∧ A «= [y].
  Proof.
    rewrite incl_lcons. intros [D E].
```

```
    apply in_sing in D. auto.
  Qed.

  Lemma incl_rcons x A B :
    A «= x::B → ¬ x el A → A «= B.

  Proof. intros C D y E. destruct (C y E) as [F|F]; congruence. Qed.

  Lemma incl_lrcons x A B :
    x::A «= x::B → ¬ x el A → A «= B.

  Proof.
    intros C D y E.
    assert (F: y el x::B) by auto.
    destruct F as [F|F]; congruence.
  Qed.

  Lemma incl_app_left A B C :
    A ++ B «= C → A «= C ∧ B «= C.
  Proof.
    firstorder.
  Qed.

End Inclusion.

Definition inclp (X : Type) (A : list X) (p : X → Prop) : Prop :=
  ∀ x, x el A → p x.
```

13.11 Setoid rewriting with list inclusion and list equivalence

```
Instance incl_preorder X :
  PreOrder (@incl X).
Proof.
  constructor; hnf; unfold incl; auto.
Qed.

Instance equi_Equivalence X :
  Equivalence (@equi X).
Proof.
  constructor; hnf; firstorder.
Qed.

Instance incl_equi_proper X :
  Proper (@equi X ==> @equi X ==> iff) (@incl X).
Proof.
  hnf. intros A B D. hnf. firstorder.
Qed.
```

```
Instance cons_incl_proper X x :
  Proper (@incl X ==> @incl X) (@cons X x).
Proof.
  hnf. apply incl_shift.
Qed.

Instance cons_equi_proper X x :
  Proper (@equi X ==> @equi X) (@cons X x).
Proof.
  hnf. firstorder.
Qed.

Instance in_incl_proper X x :
  Proper (@incl X ==> Basics.impl) (@In X x).
Proof.
  intros A B D. hnf. auto.
Qed.

Instance in_equi_proper X x :
  Proper (@equi X ==> iff) (@In X x).
Proof.
  intros A B D. firstorder.
Qed.

Instance app_incl_proper X :
  Proper (@incl X ==> @incl X ==> @incl X) (@app X).
Proof.
  intros A B D A' B' E. auto.
Qed.

Instance app_equi_proper X :
  Proper (@equi X ==> @equi X ==> @equi X) (@app X).
Proof.
  hnf. intros A B D. hnf. intros A' B' E.
  destruct D, E; auto.
Qed.
```

13.12 Equivalence

```
Section Equi.
  Variable X : Type.
  Implicit Types A B : list X.

  Lemma equi_push x A :
    x el A → A === x::A.

  Proof. auto. Qed.
```

```
  Lemma equi_dup x A :
    x::A === x::x::A.

  Proof. auto. Qed.

  Lemma equi_swap x y A:
    x::y::A === y::x::A.

  Proof. split; intros z; simpl; tauto. Qed.

  Lemma equi_shift x A B :
    x::A++B === A++x::B.

  Proof.
    split; intros y.
    - intros [D|D].
      + subst; auto.
      + apply in_app_iff in D as [D|D]; auto.
    - intros D. apply in_app_iff in D as [D|D].
      + auto.
      + destruct D; subst; auto.
  Qed.

  Lemma equi_rotate x A :
    x::A === A++[x].

  Proof.
    split; intros y; simpl.
    - intros [D|D]; subst; auto.
    - intros D. apply in_app_iff in D as [D|D].
      + auto.
      + apply in_sing in D. auto.
  Qed.
End Equi.
```

13.13 Filter

```
Definition filter (X : Type) (p : X → Prop) (p_dec : ∀ x, dec (p x)) : list X → list X :=
  fix f A := match A with
               | nil ⇒ nil
               | x::A' ⇒ if decision (p x) then x :: f A' else f A'
             end.
```

Arguments filter {X} p {p_dec} A.

```
Section FilterLemmas.
  Variable X : Type.
  Variable p : X → Prop.
```

```
Context {p_dec : ∀ x, dec (p x)}.

Lemma in_filter_iff x A :
  x el filter p A ↔ x el A ∧ p x.

Proof.
  induction A as [|y A]; simpl.
  - tauto.
  - decide (p y) as [B|B]; simpl;
    rewrite IHA; intuition; subst; tauto.
Qed.

Lemma filter_incl A :
  filter p A <<= A.

Proof.
  intros x D. apply in_filter_iff in D. apply D.
Qed.

Lemma filter_mono A B :
  A <<= B → filter p A <<= filter p B.

Proof.
  intros D x E. apply in_filter_iff in E as [E E'].
  apply in_filter_iff. auto.
Qed.

Lemma filter_id A :
  (∀ x, x el A → p x) → filter p A = A.
Proof.
  intros D.
  induction A as [|x A]; simpl.
  - reflexivity.
  - decide (p x) as [E|E].
    + f_equal; auto.
    + exfalso. auto.
Qed.

Lemma filter_app A B :
  filter p (A ++ B) = filter p A ++ filter p B.
Proof.
  induction A as [|y A]; simpl.
  - reflexivity.
  - rewrite IHA. decide (p y); reflexivity.
Qed.

Lemma filter_fst x A :
  p x → filter p (x::A) = x::filter p A.
Proof.
```

```
    simpl. decide (p x); tauto.
  Qed.

  Lemma filter_fst' x A :
    ¬ p x → filter p (x::A) = filter p A.
  Proof.
    simpl. decide (p x); tauto.
  Qed.

End FilterLemmas.

Section FilterLemmas_pq.
  Variable X : Type.
  Variable p q : X → Prop.
  Context {p_dec : ∀ x, dec (p x)}.
  Context {q_dec : ∀ x, dec (q x)}.

  Lemma filter_pq_mono A :
    (∀ x, x el A → p x → q x) → filter p A «= filter q A.

  Proof.
    intros D x E. apply in_filter_iff in E as [E E'].
    apply in_filter_iff. auto.
  Qed.

  Lemma filter_pq_eq A :
    (∀ x, x el A → (p x ↔ q x)) → filter p A = filter q A.

  Proof.
    intros C; induction A as [|x A]; simpl.
    - reflexivity.
    - decide (p x) as [D|D]; decide (q x) as [E|E].
      + f_equal. auto.
      + exfalso. apply E, (C x); auto.
      + exfalso. apply D, (C x); auto.
      + auto.
  Qed.

  Lemma filter_and A :
    filter p (filter q A) = filter (fun x ⇒ p x ∧ q x) A.
  Proof.
    set (r x := p x ∧ q x).
    induction A as [|x A]; simpl. reflexivity.
    decide (q x) as [E|E]; simpl; rewrite IHA.
    - decide (p x); decide (r x); unfold r in *|-; auto; tauto.
    - decide (r x); unfold r in *|-; auto; tauto.
  Qed.

End FilterLemmas_pq.
```

```
Section FilterComm.
  Variable X : Type.
  Variable p q : X → Prop.
  Context {p_dec : ∀ x, dec (p x)}.
  Context {q_dec : ∀ x, dec (q x)}.

  Lemma filter_comm A :
    filter p (filter q A) = filter q (filter p A).
  Proof.
    rewrite !filter_and. apply filter_pq_eq. tauto.
  Qed.
End FilterComm.
```

13.14 Element removal

```
Section Removal.
  Variable X : Type.
  Context {eq_X_dec : eq_dec X}.

  Definition rem (A : list X) (x : X) : list X :=
    filter (fun z ⇒ z ≠ x) A.

  Lemma in_rem_iff x A y :
    x el rem A y ↔ x el A ∧ x ≠ y.
  Proof.
    apply in_filter_iff.
  Qed.

  Lemma rem_not_in x y A :
    x = y ∨ ¬ x el A → ¬ x el rem A y.
  Proof.
    intros D E. apply in_rem_iff in E. tauto.
  Qed.

  Lemma rem_incl A x :
    rem A x «= A.
  Proof.
    apply filter_incl.
  Qed.

  Lemma rem_mono A B x :
    A «= B → rem A x «= rem B x.
  Proof.
    apply filter_mono.
  Qed.

  Lemma rem_cons A B x :
```

```
  A «= B → rem (x::A) x «= B.
Proof.
  intros E y F. apply E. apply in_rem_iff in F.
  destruct F as [[|]]; congruence.
Qed.

Lemma rem_cons' A B x y :
  x el B → rem A y «= B → rem (x::A) y «= B.
Proof.
  intros E F u G.
  apply in_rem_iff in G as [[[]|G] H]. exact E.
  apply F. apply in_rem_iff. auto.
Qed.

Lemma rem_in x y A :
  x el rem A y → x el A.
Proof.
  apply rem_incl.
Qed.

Lemma rem_neq x y A :
  x ≠ y → x el A → x el rem A y.
Proof.
  intros E F. apply in_rem_iff. auto.
Qed.

Lemma rem_app x A B :
  x el A → B «= A ++ rem B x.
Proof.
  intros E y F. decide (x=y) as [[]|]; auto using rem_neq.
Qed.

Lemma rem_app' x A B C :
  rem A x «= C → rem B x «= C → rem (A ++ B) x «= C.
Proof.
  unfold rem; rewrite filter_app; auto.
Qed.

Lemma rem_equi x A :
  x::A === x::rem A x.
Proof.
  split; intros y;
  intros [[]|E]; decide (x=y) as [[]|D];
  eauto using rem_in, rem_neq.
Qed.

Lemma rem_comm A x y :
  rem (rem A x) y = rem (rem A y) x.
```

```
  Proof.
    apply filter_comm.
  Qed.

  Lemma rem_fst x A :
    rem (x::A) x = rem A x.
  Proof.
    unfold rem. rewrite filter_fst'; auto.
  Qed.

  Lemma rem_fst' x y A :
    x ≠ y → rem (x::A) y = x::rem A y.
  Proof.
    intros E. unfold rem. rewrite filter_fst; auto.
  Qed.

  Lemma rem_id x A :
    ¬ x el A → rem A x = A.
  Proof.
    intros D. apply filter_id.
    intros y E F. subst. auto.
  Qed.

  Lemma rem_reorder x A :
    x el A → A === x :: rem A x.
  Proof.
    intros D. rewrite ← rem_equi. apply equi_push, D.
  Qed.

  Lemma rem_inclr A B x :
    A «= B → ¬ x el A → A «= rem B x.
  Proof.
    intros D E y F. apply in_rem_iff.
    intuition; subst; auto.
  Qed.

End Removal.

Hint Resolve rem_not_in rem_incl rem_mono rem_cons rem_cons' rem_app rem_app'
rem_in rem_neq rem_inclr.
```

13.15 Cardinality

```
Section Cardinality.
  Variable X : Type.
  Context { eq_X_dec : eq_dec X }.
  Implicit Types A B : list X.
```

```
Fixpoint card A :=
  match A with
    | nil ⇒ 0
    | x::A ⇒ if decision (x el A) then card A else 1 + card A
  end.

Lemma card_in_rem x A :
  x el A → card A = 1 + card (rem A x).
Proof.
  intros D.
  induction A as [|y A]; simpl.
  - contradiction D.
  - decide (y ≠ x) as [E|E]; simpl.
    + rewrite IHA.
      × { decide (y el A) as [G|G];
          decide (y el rem A x) as [K|K];
          (reflexivity || exfalso).
          - auto.
          - eapply G, in_rem_iff, K. }
      × destruct D; tauto.
    + apply dec_DN in E; auto; subst y.
      decide (x el A) as [F|F].
      × auto.
      × rewrite rem_id; auto.
Qed.

Lemma card_not_in_rem A x :
  ¬ x el A → card A = card (rem A x).
Proof.
  intros D; rewrite rem_id; auto.
Qed.

Lemma card_le A B :
  A «= B → card A ≤ card B.
Proof.
revert B.
induction A as [|x A]; intros B D; simpl.
- omega.
- apply incl_lcons in D as [D D1].
  decide (x el A) as [E|E].
  + auto.
  + rewrite (card_in_rem D).
    cut (card A ≤ card (rem B x)). omega.
    apply IHA. auto.
Qed.
```

```
Lemma card_eq A B :
  A === B → card A = card B.
Proof.
  intros [E F]. apply card_le in E. apply card_le in F. omega.
Qed.

Lemma card_cons_rem x A :
  card (x::A) = 1 + card (rem A x).
Proof.
  rewrite (card_eq (rem_equi x A)). simpl.
  decide (x el rem A x) as [D|D].
  - exfalso. apply in_rem_iff in D; tauto.
  - reflexivity.
Qed.

Lemma card_0 A :
  card A = 0 → A = nil.
Proof.
  destruct A as [|x A]; intros D.
  - reflexivity.
  - exfalso. rewrite card_cons_rem in D. omega.
Qed.

Lemma card_ex A B :
  card A < card B → ∃ x, x el B ∧ ¬ x el A.
Proof.
  intros D.
  decide (B «= A) as [E|E].
  - exfalso. apply card_le in E. omega.
  - apply list_exists_not_incl; auto.
Qed.

Lemma card_equi A B :
  A «= B → card A = card B → A === B.
Proof.
  revert B.
  induction A as [|x A]; simpl; intros B D E.
  - symmetry in E. apply card_0 in E. now rewrite E.
  - apply incl_lcons in D as [D D1].
    decide (x el A) as [F|F].
    + rewrite (IHA B); auto.
    + rewrite (IHA (rem B x)).
      × symmetry. apply rem_reorder, D.
      × auto.
      × apply card_in_rem in D. omega.
```

```
  Qed.

  Lemma card_lt A B x :
    A «= B → x el B → ¬ x el A → card A < card B.
  Proof.
    intros D E F.
    decide (card A = card B) as [G|G].
    + exfalso. apply F. apply (card_equi D); auto.
    + apply card_le in D. omega.
  Qed.

  Lemma card_or A B :
    A «= B → A === B ∨ card A < card B.
  Proof.
    intros D.
    decide (card A = card B) as [F|F].
    - left. apply card_equi; auto.
    - right. apply card_le in D. omega.
  Qed.

End Cardinality.

Instance card_equi_proper X (D: eq_dec X) :
  Proper (@equi X ==> eq) (@card X D).
Proof.
  hnf. apply card_eq.
Qed.
```

13.16 Duplicate-free lists

```
Inductive dupfree (X : Type) : list X → Prop :=
| dupfreeN : dupfree nil
| dupfreeC x A : ¬ x el A → dupfree A → dupfree (x::A).

Section Dupfree.
  Variable X : Type.
  Implicit Types A B : list X.

  Lemma dupfree_cons x A :
    dupfree (x::A) ↔ ¬ x el A ∧ dupfree A.
  Proof.
    split; intros D.
    - inv D; auto.
    - apply dupfreeC; tauto.
  Qed.

  Lemma dupfree_app A B :
```

```
  disjoint A B → dupfree A → dupfree B → dupfree (A++B).
Proof.
  intros D E F. induction E as [|x A E' E]; simpl.
  - exact F.
  - apply disjoint_cons in D as [D D'].
    constructor; [|exact (IHE D')].
    intros G. apply in_app_iff in G; tauto.
Qed.
Lemma dupfree_map Y A (f : X → Y) :
  (∀ x y, x el A → y el A → f x = f y → x=y) →
  dupfree A → dupfree (map f A).
Proof.
  intros D E. induction E as [|x A E' E]; simpl.
  - constructor.
  - constructor; [|now auto].
    intros F. apply in_map_iff in F as [y [F F']].
    rewrite (D y x) in F'; auto.
Qed.
Lemma dupfree_filter p (p_dec : ∀ x, dec (p x)) A :
  dupfree A → dupfree (filter p A).
Proof.
  intros D. induction D as [|x A C D]; simpl.
  - left.
  - decide (p x) as [E|E]; [|exact IHD].
    right; [|exact IHD].
    intros F. apply C. apply filter_incl in F. exact F.
 Qed.
Lemma dupfree_dec A :
  eq_dec X → dec (dupfree A).
Proof.
  intros D. induction A as [|x A].
  - left. left.
  - decide (x el A) as [E|E].
    + right. intros F. inv F; tauto.
    + destruct (IHA) as [F|F].
      × unfold dec. auto using dupfree.
      × right. intros G. inv G; tauto.
Qed.
Lemma dupfree_card A (eq_X_dec : eq_dec X) :
  dupfree A → card A = |A|.
```

```
  Proof.
    intros D.
    induction D as [|x A E F]; simpl.
    - reflexivity.
    - decide (x el A) as [G|].
      + contradiction (E G).
      + omega.
  Qed.
End Dupfree.
Section Undup.
  Variable X : Type.
  Context {eq_X_dec : eq_dec X}.
  Implicit Types A B : list X.
  Fixpoint undup (A : list X) : list X :=
    match A with
      | nil ⇒ nil
      | x::A' ⇒ if decision (x el A') then undup A' else x :: undup A'
    end.
  Lemma undup_id_equi A :
    undup A === A.
  Proof.
    induction A as [|x A]; simpl.
    - reflexivity.
    - decide (x el A) as [E|E]; rewrite IHA; auto.
  Qed.
  Lemma dupfree_undup A :
    dupfree (undup A).
  Proof.
    induction A as [|x A]; simpl.
    - left.
    - decide (x el A) as [E|E]; auto.
      right; auto. now rewrite undup_id_equi.
  Qed.
  Lemma undup_incl A B :
    A «= B ↔ undup A «= undup B.
  Proof.
    now rewrite !undup_id_equi.
  Qed.
  Lemma undup_equi A B :
    A === B ↔ undup A === undup B.
  Proof.
```

```
    now rewrite !undup_id_equi.
  Qed.

  Lemma undup_id A :
    dupfree A → undup A = A.
  Proof.
    intros E. induction E as [|x A E F]; simpl.
    - reflexivity.
    - rewrite IHF. decide (x el A) as [G|G]; tauto.
  Qed.

  Lemma undup_idempotent A :
    undup (undup A) = undup A.

  Proof. apply undup_id, dupfree_undup. Qed.

End Undup.
```

13.17 Power lists

```
Section PowerRep.
  Variable X : Type.
  Context {eq_X_dec : eq_dec X}.

  Fixpoint power (U : list X ) : list (list X) :=
    match U with
      | nil ⇒ [nil]
      | x :: U' ⇒ power U' ++ map (cons x) (power U')
    end.

  Lemma power_incl A U :
    A el power U → A «= U.

  Proof.
    revert A; induction U as [|x U]; simpl; intros A D.
    - destruct D as [[]|[]]; auto.
    - apply in_app_iff in D as [E|E]. now auto.
      apply in_map_iff in E as [A' [E F]]. subst A.
      auto.
  Qed.

  Lemma power_nil U :
    nil el power U.

  Proof. induction U; simpl; auto. Qed.

  Definition rep (A U : list X) : list X :=
    filter (fun x ⇒ x el A) U.

  Lemma rep_power A U :
```

rep A U el power U.

Proof.
revert A; induction U as [|x U]; intros A.
- simpl; auto.
- simpl. *decide* (x el A) as [D|D]; auto using in_map.
Qed.

Lemma rep_incl A U :
rep A U ≪= A.

Proof.
unfold rep. intros x D. apply in_filter_iff in D. apply D.
Qed.

Lemma rep_in x A U :
A ≪= U → x el A → x el rep A U.
Proof.
intros D E. apply in_filter_iff; auto.
Qed.

Lemma rep_equi A U :
A ≪= U → rep A U === A.

Proof.
intros D. split. *now* apply rep_incl.
intros x. apply rep_in, D.
Qed.

Lemma rep_mono A B U :
A ≪= B → rep A U ≪= rep B U.

Proof. intros D. apply filter_pq_mono. auto. Qed.

Lemma rep_eq' A B U :
(∀ x, x el U → (x el A ↔ x el B)) → rep A U = rep B U.

Proof. intros D. apply filter_pq_eq. auto. Qed.

Lemma rep_eq A B U :
A === B → rep A U = rep B U.

Proof. intros D. apply filter_pq_eq. firstorder. Qed.

Lemma rep_injective A B U :
A ≪= U → B ≪= U → rep A U = rep B U → A === B.

Proof.
intros D E F. transitivity (rep A U).
- symmetry. apply rep_equi, D.
- rewrite F. apply rep_equi, E.
Qed.

```
Lemma rep_idempotent A U :
  rep (rep A U) U = rep A U.
Proof.
  unfold rep at 1 3. apply filter_pq_eq.
  intros x D. split.
  + apply rep_incl.
  + intros E. apply in_filter_iff. auto.
Qed.
Lemma dupfree_power U :
  dupfree U → dupfree (power U).
Proof.
  intros D. induction D as [|x U E D]; simpl.
  - constructor. now auto. constructor.
  - apply dupfree_app.
    + intros [A [F G]]. apply in_map_iff in G as [A' [G G']].
      subst A. apply E. apply (power_incl F). auto.
    + exact IHD.
    + apply dupfree_map; congruence.
Qed.
Lemma dupfree_in_power U A :
  A el power U → dupfree U → dupfree A.
Proof.
  intros E D. revert A E.
  induction D as [|x U D D']; simpl; intros A E.
  - destruct E as [[]|[]]. constructor.
  - apply in_app_iff in E as [E|E].
    + auto.
    + apply in_map_iff in E as [A' [E E']]. subst A.
      constructor.
      × intros F; apply D. apply (power_incl E'), F.
      × auto.
Qed.
Lemma rep_dupfree A U :
  dupfree U → A el power U → rep A U = A.
Proof.
  intros D; revert A.
  induction D as [|x U E F]; intros A G.
  - destruct G as [[]|[]]; reflexivity.
  - simpl in G. apply in_app_iff in G as [G|G].
    + simpl. decide (x el A) as [H|H].
      × exfalso. apply E. apply (power_incl G), H.
```

```
      × auto.
    + apply in_map_iff in G as [A' [G H]]. subst A.
      simpl. decide (x=x ∨ x el A') as [G|G].
      × f_equal. rewrite ← (IHF A' H) at 2.
        apply filter_pq_eq. apply power_incl in H.
        intros z K. split; [|now auto].
        intros [L|L]; subst; tauto.
      × exfalso; auto.
  Qed.

  Lemma power_extensional A B U :
    dupfree U → A el power U → B el power U → A === B → A = B.

  Proof.
    intros D E F G.
    rewrite ← (rep_dupfree D E). rewrite ← (rep_dupfree D F).
    apply rep_eq, G.
  Qed.

End PowerRep.
```

13.18 Finite closure iteration

```
Module FCI.
Section FCI.
  Variable X : Type.
  Context {eq_X_dec : eq_dec X}.
  Variable step : list X → X → Prop.
  Context {step_dec : ∀ A x, dec (step A x)}.
  Variable V : list X.

  Lemma pick (A : list X) :
    { x | x el V ∧ step A x ∧ ¬ x el A } + { ∀ x, x el V → step A x → x el A }.
  Proof.
    destruct (list_sigma_forall V (fun x ⇒ step A x ∧ ¬ x el A)) as [E|E].
    - auto.
    - right. intros x F G.
      decide (x el A). assumption. exfalso.
      eapply E; eauto.
  Qed.

  Definition F (A : list X) : list X.
    destruct (pick A) as [[x _]|_].
    - exact (x::A).
    - exact A.
```

```
Defined.

Definition C := it F (card V) nil.

Lemma it_incl n :
  it F n nil <<= V.
Proof.
  apply it_ind. now auto.
  intros A E. unfold F.
  destruct (pick A) as [[x G]|G]; intuition.
Qed.

Lemma incl :
  C <<= V.
Proof.
  apply it_incl.
Qed.

Lemma ind p :
  (∀ A x, inclp A p → x el V → step A x → p x) → inclp C p.
Proof.
  intros B. unfold C. apply it_ind.
  + intros x [].
  + intros D G x. unfold F.
    destruct (pick D) as [[y E]|E].
    × intros [[]|]; intuition; eauto.
    × intuition.
Qed.

Lemma fp :
  F C = C.
Proof.
  pose (sigma (A : list X) := card V - card A).
  replace C with (it F (sigma nil) nil).
  - apply it_fp. intros n. simpl.
    set (J:= it F n nil). unfold FP, F.
    destruct (pick J) as [[x B]|B].
    + right.
      assert (G: card J < card (x :: J)).
      { apply card_lt with (x:=x); intuition. }
      assert (H: card (x :: J) ≤ card V).
      { apply card_le, incl_cons. apply B. apply it_incl. }
      unfold sigma. omega.
    + auto.
  - unfold C, sigma. f_equal. change (card nil) with 0. omega.
Qed.
```

```
  Lemma closure x :
    x el V → step C x → x el C.
  Proof.
    assert (A2:= fp).
    unfold F in A2.
    destruct (pick C) as [[y _]| B].
    + contradiction (list_cycle A2).
    + apply B.
  Qed.
End FCI.
End FCI.
```

13.19 Deprecated names, defined for backward compatibilitly

```
Definition dupfree_inv := dupfree_cons.
```