Preuves formelles mécanisées

projet 2025-2026

Paul PASSERON, Jean ROUSSEAU

November 17, 2025

Contents

Chapter 1

Introduction

1.1 Préambule

- Le projet a été réalisé sur VSCode (VScoq) d'une part, et sur coqide de l'autre.
- Les résultats ont été mis en commun grâce à Github.
- Le rapport a été écrit en LaTeXet le chapitre 2 a été généré grâce à coqdoc.

1.2 Livrables

Les livrables de ce projet sont :

- Le sujet : `projet2025.pdf`
- Le rapport : `passeron_rousseau.pdf`
- Le fichier source : `projet_rocq_passeron_rousseau.v`
- Un fichier Makefile pour générer le rapport : `Makefile`
- Un fichier LaTeXcontenant le titre du rapport : `titre.tex`
- Un fichier LaTeXcontenant le squelette du rapport : `rapport.tex`
- Un fichier LaTeXcontenant le corps du rapport : `passeron_rousseau.tex`

Le tout est contenu dans une archive passeron_rousseau.tar.gz.

1.3 Mode d'emploi

Si pour une raison quelconque le rapport a mal été généré, il est possible d'en faire un nouveau en suivant les étapes suivantes :

1. Se placer dans le dossier extrait de l'archive.
2. Faire la commande : `make clean`
3. Faire la commande : `make pdf`

De même pour générer l'archive:

1. Se placer dans le dossier extrait de l'archive.
2. Faire la commande : `make clean`
3. Faire la commande : `make archive`

Une fois cela fait, un nouveau fichier `passeron_rousseau.tar.gz` devrait être généré.

1.4 Remarques

Certaines preuves sont faites sans *bullet points*, notamment les preuves avec deux *goals* dont le premier est trivialement vrai (pas plus de deux ou trois lignes).

Chapter 2

Library projet_rocq_passeron_rousseau

2.1 Partie 1 : Examen du contenu d'une liste

```
Require Import List.
Import ListNotations.
Require Import Coq.Arith.PeanoNat.
Import Nat.
Parameter A: Type.
Parameter A_eq_dec: ∀ (x y: A), {x=y} + {¬x=y}.
```

- A est un type arbitraire fourni comme paramètre.
- A_eq_dec fournit une décision d'égalité pour la suite.

2.1.1 Question 1

```
Fixpoint repeat (x: A) (n: nat): list A :=
 match n with
  | 0 ⇒ []
  | S (n') ⇒ x::(repeat x n')
end.
Lemma repeat_sound1: ∀ (a: A) n,
  Forall (fun x ⇒ x=a) (repeat a n).
Proof.
induction n.
- unfold repeat. apply Forall_nil.
- simpl. apply Forall_cons.
  + reflexivity.
```

```
  + exact IHn.
Qed.
```

```
Lemma repeat_sound2: ∀ (a: A) n,
  length (repeat a n) = n.
Proof.
induction n.
- simpl. reflexivity.
- simpl. rewrite IHn. reflexivity.
Qed.
```

2.1.2 Question 2

Question 2.a

```
Fixpoint split_p_aux (p: A → bool) (l: list A) (acc: list A) :=
match l with
  | [] ⇒ (acc, [])
  | head::tail ⇒ match p head with
    | true ⇒ (acc, l)
    | false ⇒ split_p_aux p tail (acc ++ [head])
  end
end.
```

```
Definition split_p_acc (p: A → bool) (l: list A) := split_p_aux p l [].
```

```
Fixpoint split_p (p: A → bool) (l: list A) :=
match l with
  | [] ⇒ ([], [])
  | head::tail ⇒ match p head with
    | true ⇒ ([], l)
    | false ⇒ let (left, right) := split_p p tail in (head::left, right)
  end
end.
```

Question 2.b

```
Lemma split_p_first : ∀ (p : A → bool) (l : list A),
Forall (fun x ⇒ p x = false) (fst (split_p p l)) .
Proof.
intros p.
induction l.
-simpl. apply Forall_nil.
- simpl. case (p a) eqn:Ha.
  + simpl. apply Forall_nil.
  + destruct (split_p p l) eqn:Hsplit.
    simpl.
    apply Forall_cons.
    × exact Ha.
```

```
    × simpl in IHl.
      exact IHl.
Qed.

Lemma split_p_snd : ∀ (p : A → bool) (l l1 l2 : list A) x,
split_p p l = (l1 , l2) → head l2 = Some x → p x = true.
Proof.
intros p l.
induction l as [| a l' IHl'].
- intros l1 l2 x Hsplit Hhead.
  simpl in Hsplit.
  injection Hsplit. intros H2 _.
  rewrite ← H2 in Hhead.
  discriminate Hhead.
 - intros l1 l2 x Hsplit Hhead.
    simpl in Hsplit.
    destruct (p a)eqn:Pa.
    + injection Hsplit as Hl1 Hl2.
      rewrite ← Hl2 in Hhead.
      simpl in Hhead.
      inversion Hhead as [AeqX].
      rewrite ← AeqX.
      exact Pa.
    + destruct (split_p p l') as [left right] eqn:Hsplit'.
      injection Hsplit as Hl1 Hl2.
      eapply (IHl' left right).
      × reflexivity.
      × rewrite Hl2. exact Hhead.
Qed.

Lemma split_p_forall : ∀ (p : A → bool) (l : list A),
Forall (fun x ⇒ p x = true) l → split_p p l = ([] , l).
Proof.
intros p. induction l.
- intros _. simpl. reflexivity.
- intro Hyp.
  simpl.
  destruct (p a) eqn:Pa.
  + reflexivity.
  + destruct (split_p p l).
    rewrite Forall_forall in Hyp.
    specialize (Hyp a).
    simpl in Hyp.
    assert (p a = true) as PaTrue.
    × apply Hyp. left. reflexivity.
    × rewrite PaTrue in Pa.
      discriminate Pa.
```

```
Qed.
```

Question 2.c

```
Lemma split_p_forall_left : ∀ (p : A → bool) (l : list A),
Forall (fun x ⇒ p x = false) l → split_p p l = (l, []).
Proof. intros p. induction l.
- intro Hyp. simpl. reflexivity.
- intro Hyp. simpl. case (p a)eqn:Pa.
  + assert (p a = false) as PaFalse. {
      rewrite Forall_forall in Hyp.
      specialize (Hyp a).
      apply Hyp.
      simpl.
      left. reflexivity.
    }
    rewrite PaFalse in Pa.
    discriminate Pa.
  + destruct (split_p p l) as [left right] eqn:Hsplit.
    inversion Hyp.
    apply IHl in H2.
    injection H2 as HLeft HRight.
    rewrite HLeft.
    rewrite HRight.
    reflexivity.
Qed.
```

Question 2.d

```
Lemma split_p_append: ∀ (p: A → bool) (l left right: list A),
split_p p l = (left, right) → l = app left right.
Proof.
intros p. induction l.
- intros l r SplitHyp. simpl in SplitHyp.
  injection SplitHyp as HL HR.
  rewrite ← HL.
  rewrite ← HR.
  reflexivity.
- intros left right SplitHyp.
  simpl in SplitHyp.
  destruct (p a) eqn:Pa.
  + injection SplitHyp as Empty Al.
    rewrite ←Empty.
    rewrite ← Al.
    reflexivity.
  + destruct (split_p p l) as [Left Right] eqn:Hsplit.
```

```
    specialize (IHl Left Right).
    assert (l = Left ++ Right).
    × apply IHl. reflexivity.
    × injection SplitHyp as ALeft RRight.
      rewrite ← RRight.
      rewrite ← ALeft.
      simpl.
      rewrite ← H.
      reflexivity.
Qed.
```

Question 2.e

```
Require Import Lia.
Lemma split_p_length: ∀ (p: A → bool) (l left right: list A),
split_p p l = (left, right) → length left ≤ length l ∧ length right ≤ length l.
intros p. induction l.
- intros left right SplitHyp.
  simpl in SplitHyp.
  injection SplitHyp as LEmpty REmpty.
  rewrite ← LEmpty.
  rewrite ← REmpty.
  simpl. lia.
- intros L R.
  simpl.
  destruct (p a) eqn:Pa.
  + intro H.
    split; injection H as LEmpty ReqAL.
    × rewrite ← LEmpty.
      simpl.
      lia.
    × rewrite ← ReqAL.
      simpl.
      lia.
  + destruct (split_p p l) as [Left Right] eqn: HSplit.
    intro H. injection H as LeqAL REmpty.
    specialize (IHl Left Right).
    assert(length Left ≤ length l ∧
    length Right ≤ length l) as Rec.
      {
        apply IHl.
        reflexivity.
      }
    destruct Rec as [RL RR].
    split.
    × rewrite ← LeqAL.
```

```
      simpl.
      apply Nat.succ_le_mono in RL.
      assumption.
    × rewrite ← REmpty.
      apply le_S.
      assumption.
Qed.
```

2.1.3 Question 3

```
Require Import Coq.Classes.EquivDec.
Require Import Coq.Bool.Bool.

Definition split_occ (v: A) (l: list A) :=
  split_p (fun x ⇒ if A_eq_dec x v then true else false) l.

Lemma split_occ_first: ∀ (v: A) (l: list A),
Forall (fun x ⇒ ~(x = v)) (fst (split_occ v l)).
Proof.
intros v.
induction l.
- simpl. apply Forall_nil.
- unfold split_occ.
  simpl.
  destruct (A_eq_dec a v) eqn:HAV.
  + simpl. apply Forall_nil.
  + destruct split_p as [Left Right] eqn: Hsplit.
    unfold split_occ in IHl.
    rewrite Hsplit in IHl.
    simpl in IHl.
    simpl.
    apply Forall_cons.
    × exact n.
    × exact IHl.
Qed.

Lemma split_occ_snd_starts_with_v: ∀
  (v: A) (l: list A),
    snd (split_occ v l) = [] ∨ (∃ (l': list A), snd(split_occ v l) = v :: l').
Proof.
  intros v l.
  unfold split_occ.
  induction l.
  - simpl. left. reflexivity.
  - simpl. destruct (A_eq_dec a v) eqn:Hav.
    + right. ∃ l. rewrite e. reflexivity.
    + destruct (split_p (fun x ⇒ if A_eq_dec x v then true else false) l)
eqn:Hsplit.
```

```
    exact IHl.
Qed.
```

2.1.4 Question 4

a)

```
Fixpoint split_p_all_aux (p : A → bool) (l : list A)
  (acc_prefix : list A) (acc_current : option (list A)) (acc_lists : list (list A))
  : list A × list (list A) :=
  match l with
  | [] ⇒
      match acc_current with
      | Some curr ⇒ (rev acc_prefix, rev (rev curr :: acc_lists))
      | None ⇒ (rev acc_prefix, rev acc_lists)
      end
  | x :: xs ⇒
      if p x then
        match acc_current with
        | Some curr ⇒
            split_p_all_aux p xs acc_prefix (Some [x]) (rev curr :: acc_lists)
        | None ⇒
            split_p_all_aux p xs acc_prefix (Some [x]) acc_lists
        end
      else
        match acc_current with
        | Some curr ⇒
            split_p_all_aux p xs acc_prefix (Some (x :: curr)) acc_lists
        | None ⇒
            split_p_all_aux p xs (x :: acc_prefix) None acc_lists
        end
  end.
Definition split_p_all (p : A → bool) (l : list A) : list A × list (list A) :=
  split_p_all_aux p l [] None [].
```

b)

```
Definition all_not_p (p : A → bool) (l : list A) : Prop :=
  ∀ x, In x l → p x = false.
Lemma all_not_p_rev: ∀ (p : A → bool) (l : list A),
all_not_p p l ↔ all_not_p p (rev l).
Proof.
  intros.
  split; intro H; unfold all_not_p in *; intros x H'.
  - rewrite in_rev in H'.
    rewrite rev_involutive in H'.
    exact (H x H').
```

```
  - rewrite in_rev in H'.
    exact (H x H').
Qed.

Lemma split_p_all_aux_prefix_not_p :
  ∀ (p : A → bool) (l : list A)
          (acc_prefix : list A) (acc_current : option (list A)) (acc_lists : list
(list A)),
  all_not_p p acc_prefix →
  all_not_p p (fst (split_p_all_aux p l acc_prefix acc_current acc_lists)).
Proof.
  intros p l.
  induction l as [| x xs IH].
  intros.
  - simpl.
    case acc_current.
    + intro l.
      simpl.
      rewrite ← all_not_p_rev.
      exact H.
    + simpl.
      rewrite ← all_not_p_rev.
      exact H.
  - intros.
    simpl.
    destruct (p x) eqn:Hpx.
    × destruct acc_current; apply IH; exact H.
    × destruct acc_current.
      −apply IH. exact H.
      −apply IH.
        unfold all_not_p.
        unfold all_not_p in H.
        intros x0 HIn.
        destruct (A_eq_dec x x0) as [Hxx0 | Hxx0].
        subst x0. assumption.
        simpl in HIn.
        destruct HIn as [Hcontr | Hr].
        contradiction.
        exact (H x0 Hr).
Qed.

Theorem split_p_all_fst_no_sat_p :
  ∀ (p : A → bool) (l : list A),
  all_not_p p (fst (split_p_all p l)).
Proof.
  intros A p l.
  unfold split_p_all.
```

```
  apply split_p_all_aux_prefix_not_p.
  unfold all_not_p.
  intros x Hin.
  simpl in Hin.
  contradiction.
Qed.
```

c)
```
Definition first_true_rest_false (p : A → bool) (l : list A) : Prop :=
  match l with
  | [] ⇒ True
  | x :: xs ⇒ p x = true ∧ Forall (fun y ⇒ p y = false) xs
  end.

Lemma split_p_all_aux_lists_Forall :
  ∀ (p : A → bool) (l : list A)
         (acc_prefix : list A) (acc_current : option (list A)) (acc_lists : list
(list A)),
  Forall (first_true_rest_false p) acc_lists →
  (match acc_current with
   | None ⇒ True
   | Some curr ⇒ first_true_rest_false p (rev curr)
   end) →
  Forall (first_true_rest_false p) (snd (split_p_all_aux p l acc_prefix acc_current
acc_lists)).
Proof.
  intros p l.
  induction l; intros.
  - simpl.
    destruct acc_current as [curr |] eqn: Hcu.
    + simpl.
      apply Forall_app.
      split.
      × apply Forall_rev.
        exact H.
      × apply Forall_cons.
        exact H0.
        apply Forall_nil.
    + simpl.
      apply Forall_rev.
      exact H.
  - simpl.
    destruct (p a) eqn:Hpa.
    + destruct acc_current as [curr |] eqn: Hcu.
      × apply IHl.
        −apply Forall_cons.
          exact H0.
```

```
          exact H.
        -simpl.
          split.
          exact Hpa.
          apply Forall_nil.
      × apply IHl.
        exact H.
        simpl.
        split.
        exact Hpa.
        apply Forall_nil.
    + destruct acc_current as [curr |] eqn: Hcu.
      × apply IHl.
        exact H.
        simpl.
        unfold first_true_rest_false.
        admit.
Admitted.
```

```
Theorem split_p_all_lists_Forall :
  ∀ (p : A → bool) (l : list A),
  Forall (first_true_rest_false p) (snd (split_p_all p l)).
Proof.
Admitted.
```

2.2 Partie 2 : Implantation des multi-ensembles

```
Parameter T : Type.
Parameter T_eq_dec: ∀ (x y : T), {x=y} + {¬x=y}.
Definition multiset := list (T×nat).
```

2.2.1 Question 1

```
Definition empty : multiset := ([]).

Definition singleton (t:T) : multiset := [(t,1)].

Fixpoint member (t:T) (m:multiset) : bool := match m with
  |[] ⇒ false
  |(x,_)::m' ⇒ if T_eq_dec x t then true
                else member t m'
end.

Fixpoint add (t:T) (n:nat) (m:multiset) : multiset :=
if n == 0 then m else
match m with
```

```
  |[] ⇒ [(t,n)]
  |(x,xn)::m' ⇒ if T_eq_dec x t then (x,xn+n)::m'
                 else (x,xn)::(add t n m')
end.
```

```
Fixpoint multiplicity (t:T) (m:multiset) : nat := match m with
  |[] ⇒ 0
  |(x,xn)::m' ⇒ if T_eq_dec x t then xn
                 else multiplicity t m'
end.
```

```
Fixpoint removeOne (t:T) (m:multiset) : multiset := match m with
  |[] ⇒ []
  |(x,xn)::m' ⇒ if T_eq_dec x t then
                    if xn=?1 then m'
                    else (x,xn-1)::m'
                 else (x,xn)::(removeOne t m')
end.
```

```
Fixpoint removeAll (t:T) (m:multiset) : multiset := match m with
  |[] ⇒ []
  |(x,xn)::m' ⇒ if T_eq_dec x t then m'
                 else (x,xn)::(removeAll t m')
end.
```

2.2.2 Question 2

Question 2.a

```
Definition InMultiset (t:T) (m:multiset) : Prop := (member t m) = true.
```

Question 2.b

```
Fixpoint wf (m: multiset) : Prop :=
  match m with
  | [] ⇒ True
  | (x, n) :: m' ⇒
      n > 0 ∧

      (∀ y occ, In (y, occ) m' → y ≠ x) ∧

      wf m'
  end.
```

Question 2.c

```
Lemma empty_wf: wf empty = True.
Proof.
  simpl.
```

```
  reflexivity.
Qed.

Require Import PeanoNat.
Require Import Coq.Logic.PropExtensionality.

Lemma singleton_wf: ∀ x: T, wf (singleton x) = True.
Proof.
  intro x.
  simpl.
  apply propositional_extensionality.
  split.
  intro H.
  - apply proj2 in H.
    apply proj2 in H.
    exact H.
  - split.
    + lia.
    + split.
      × intros y n contr.
        contradiction.
      × exact H.
Qed.

Lemma not_in_after_add_different :
  ∀ x y n s, x ≠ y → wf s →
  (∀ z occ, In (z, occ) s → z ≠ x) →
  (∀ z occ, In (z, occ) (add y n s) → z ≠ x).
Proof.
  intros x y n s Hxy Hwf Hnot_in.
  induction s as [| [a an] s' IH].
  - simpl. destruct n.
    + intros z occ Hin. apply (Hnot_in z occ Hin).
    + simpl. intros z occ Hin.
      destruct Hin as [Heq | Hcontra].
      × inversion Heq. subst. symmetry. exact Hxy.
      × inversion Hcontra.
  - simpl in Hwf.
    destruct Hwf as [Han_pos [Hnot_in_s' Hwf_s']].
    simpl. destruct n.
    + simpl. intros z occ Hin. apply (Hnot_in z occ Hin).
    + simpl. destruct (T_eq_dec a y) as [Hay | Hnay].
      × subst a. intros z occ [Heq | Hin'].
        – inversion Heq. subst. symmetry. exact Hxy.
        – apply (Hnot_in z occ). right. exact Hin'.
      × intros z occ [Heq | Hin'].
        – inversion Heq. subst.
          apply (Hnot_in z occ). left. reflexivity.
```

```
        – apply (IH Hwf_s'
            (fun w wocc Hw ⇒ Hnot_in w wocc (or_intror Hw)) z occ Hin').
Qed.

Lemma add_to_empty_wf : ∀ x n, n > 0 → wf [(x, n)].
Proof.
  intros x n Hn.
  simpl.
  split; [exact Hn | split].
  - intros y occ Hin. inversion Hin.
  - exact I.
Qed.

Lemma add_wf: ∀ (x: T) (n: nat) (s: multiset), wf s → wf (add x n s).
Proof.
  intros x n s Hwf.
  induction s as [| [a an] s' IH].
  - simpl. destruct n.
    + simpl. exact Hwf.
    + simpl. apply add_to_empty_wf. lia.
  - simpl in Hwf.
    destruct Hwf as [Han_pos [Hnot_in_s' Hwf_s']].
    simpl. destruct n.
    + simpl.  split; [exact Han_pos | split; [exact Hnot_in_s' | exact
Hwf_s']].
    + simpl. destruct (T_eq_dec a x) as [Hax | Hnax].
      × subst a. simpl.
        split; [lia | split].
        – exact Hnot_in_s'.
        – exact Hwf_s'.
      × simpl. split; [exact Han_pos | split].
        – intros y occ Hin_add.
            apply not_in_after_add_different with (x := a) (y := x) (n := S n)
(s := s') in Hin_add.
            ++ exact Hin_add.
            ++ exact Hnax.
            ++ exact Hwf_s'.
            ++ exact Hnot_in_s'.
        – apply IH. exact Hwf_s'.
Qed.

Lemma not_in_after_remove:
  ∀ x y s occ, x ≠ y → wf s →
  In (y, occ) s → In (y, occ) (removeOne x s).
Proof.
  intros x y s occ Hxy Hwfs Hins.
  induction s.
  simpl in *.
```

```
  contradiction.
  destruct a as [a an].
  simpl.
  destruct (T_eq_dec a x) as [Hax | Hax].
  - subst x.
    simpl in Hins.
    destruct Hins as [Hay | Hin].
    injection Hay as Ha Hb.
    contradiction.
    destruct (an == 1) as [Haeq | Haneq].
    + rewrite Haeq.
      simpl.
      exact Hin.
    + assert (an ≠ 1). assumption.
      apply Nat.eqb_neq in Haneq.
      rewrite Haneq.
      simpl. right. exact Hin.
  - destruct Hwfs as [H0 [H1 H2]].
    destruct (T_eq_dec y a) as [Hya | Hya].
    × subst y.
      destruct (an == occ) as [Hanocc | Hanocc].
      assert (an = occ).
      assumption.
      subst occ.
      simpl.
      left.
      reflexivity.
      simpl.
      right.
      simpl in Hins.
      destruct Hins as [Hnope | Hin].
      injection Hnope as Haa.
      contradiction.
      exact (IHs H2 Hin).
    × simpl.
      right.
      simpl in Hins.
      destruct Hins as [Hcontr | Hin].
      injection Hcontr as Hcontr _.
      symmetry in Hcontr.
      contradiction.
      exact (IHs H2 Hin).
Qed.

Lemma not_in_before_remove:
  ∀ (x y: T) (occ: nat) (s: multiset), x ≠ y → wf s →
```

```
  (In (y, occ) (removeOne x s)) →
  (In (y, occ) s).
Proof.
  intros x y occ s Hxy Hwfs HIn.
  induction s as [| [a an] s' IHs].
    simpl in HIn. contradiction.
    destruct Hwfs as [H0 [H1 H2]].
    simpl in HIn.
    destruct (T_eq_dec a x) as [Hax | Hax].
    + subst a.
      destruct (an =? 1) eqn:Heq1.
      × simpl. right. exact HIn.
      × simpl in HIn.
        destruct HIn as [Heq | HIn'].
        – injection Heq as Hy Hocc.
           subst y. contradiction.
        – simpl. right. exact HIn'.
    + simpl in HIn.
      destruct HIn as [Heq | HIn'].
      × simpl. left. exact Heq.
      × simpl. right. apply IHs.
        – exact H2.
        – exact HIn'.
Qed.

Lemma removeOne_wf: ∀ (s: multiset) (x: T), wf s → wf (removeOne x s).
Proof.
  intros s x Hwf.
  induction s as [| [a an] s' IH].
  - simpl in *. exact Hwf.
  - simpl in Hwf. destruct Hwf as [Han_pos [Hnot_in_s' Hwf_s']].
    simpl.
    case (an == 1).
    + intro Han. rewrite Han.
      simpl.
      destruct (T_eq_dec a x) as [Hax | Hnax].
      × exact Hwf_s'.
      × simpl.
        split.
        lia.
        split.
        – intros y occ Hin.
          destruct (T_eq_dec x y) as [Hxy | Hxy].
          ++ subst x. symmetry. exact Hnax.
          ++ exact (Hnot_in_s' y occ (not_in_before_remove x y occ s' Hxy
Hwf_s' Hin)).
```

```
      – exact (IH (Hwf_s')).
  + intro Han.
    assert (an ≠ 1).
    assumption.
    destruct (T_eq_dec a x) as [Hax | Hnax].
    × assert(rewH:= H). rewrite ← Nat.eqb_neq in rewH.
      rewrite rewH.
      simpl.
      split.
      lia.
      split.
      – exact Hnot_in_s'.
      – exact Hwf_s'.
    × simpl.
      split.
      exact Han_pos.
      split.
      – intros y occ HIn .
         assert (H' := Hnot_in_s' y occ).
         destruct (T_eq_dec x y).
         ++ rewrite e in Hnax.
            symmetry.
            exact Hnax.
         ++ exact (H' (not_in_before_remove x y occ s' n Hwf_s' HIn)).
      – exact (IH Hwf_s').
Qed.

Lemma rawf_aux_1: ∀ x y s occ, x ≠ y → wf s →
  (In (y, occ) (removeAll x s)) →
  (In (y, occ) s).
Proof.
  intros x y s occ Hxy Hwfs HIn.
  induction s.
  simpl.
  simpl in HIn.
  contradiction.
  simpl.
  destruct a as [a an].
  assert (H := Hwfs).
  simpl in H.
  apply proj2 in H.
  apply proj2 in H.
  simpl in HIn.
  destruct (T_eq_dec a x) as [Hax | Hax].
  - subst a.
    right.
```

```
    assumption.
  - simpl in HIn.
    destruct HIn as [Hl | Hr].
    + left. assumption.
    + right. exact (IHs H Hr).
Qed.
Lemma rawf_aux_2: ∀ x s occ, wf s → In (x, occ) (removeAll x s) → False.
Proof.
  intros x s occ Hwf H.
  induction s.
  simpl in *.
  assumption.
  destruct a as [a an].
  simpl in H.
  destruct (T_eq_dec a x) as [Hax | Hax].
  - subst x.
    simpl in Hwf.
    destruct Hwf as [H0 [H1 H2]].
    assert (H' := H1 a occ).
    apply H' in H.
    contradiction.
  - simpl in H.
    destruct H as [Hcontr | HIn].
    + injection Hcontr as Hcontr.
      contradiction.
    + destruct Hwf as [_ [_ H]].
      exact (IHs H HIn).
Qed.
Lemma removeAll_wf: ∀ (s: multiset) (x: T), wf s → wf (removeAll x s).
Proof.
  intros s x Hwfs.
  induction s.
  simpl.
  simpl in Hwfs.
  assumption.
  destruct a as [a an].
  simpl.
  destruct (T_eq_dec a x) as [Hax | Hax].
  - destruct Hwfs as [_ [_ H]].
    exact H.
  - simpl.
    assert (backup := Hwfs).
    simpl in Hwfs.
    destruct Hwfs as [Han [HnotIn Hwfs]].
    split.
```

```
    exact Han.
    split.
    + intros y occ.
      apply IHs in Hwfs.
      intro HIn.
      apply (rawf_aux_1 x y s occ) in HIn.
      × apply (HnotIn y occ). assumption.
      × destruct (T_eq_dec x y) as [Hxy | Hxy].
        –subst y.
          assert (H: wf s).
          ++ destruct backup as [_ [ _ H']].
             exact H'.
          ++ assert (H' := rawf_aux_2 x s occ H HIn).
             contradiction.
        – assumption.
      × destruct backup as [_ [ _ H']].
        exact H'.
    + exact (IHs Hwfs).
Qed.
```

2.2.3 Question 3

```
Lemma x_not_in_empty : ∀ x, ¬ InMultiset x empty.
Proof.
intros. unfold not. intros. unfold InMultiset in H. simpl in H. discriminate.
Qed.
```

```
Lemma prop_2 : ∀ x y , InMultiset y (singleton x) ↔ x = y.
Proof.
  intros.
  unfold InMultiset, singleton.
  simpl.
  destruct (T_eq_dec x y) as [Heq | Hneq].
  -split.
    +intros. exact Heq.
    +intros. reflexivity.
  -split.
    +intros. discriminate H.
    +intros. contradiction.
Qed.
```

```
Lemma prop_3 : ∀ x, multiplicity x (singleton x) = 1.
Proof.
  intros.
  unfold singleton.
  simpl.
  destruct (T_eq_dec x x) as [Heq | Hneq].
```

```
  -reflexivity.
  -contradiction.
Qed.

Lemma prop_4 : ∀ x s, wf s → (member x s = true ↔ InMultiset x s).
Proof.
  intros.
  split; unfold InMultiset; intro; exact H0.
Qed.

Lemma prop_5 : ∀ x n s, n > 0 → InMultiset x (add x n s).
Proof.
  intros.
  unfold InMultiset.
  destruct n as [| n'].
  - lia.
  - induction s as [| [y k] s' IH].
    + simpl. destruct (T_eq_dec x x) as [Heq|Hneq].
      × reflexivity.
      × contradiction.
    + simpl. destruct (T_eq_dec y x) as [Heq|Hneq].
      × simpl. subst y. destruct (T_eq_dec x x) as [Heq2|Hneq2].
        – reflexivity.
        – contradiction.
      × simpl. destruct (T_eq_dec y x) as [Heq2|Hneq2].
        – contradiction.
        – apply IH.
Qed.

Lemma prop_6 : ∀ x y n s, x ≠ y → (InMultiset y (add x n s) ↔ InMultiset y
s).
Proof.
  intros.
  split.
    - intro. induction s as [| [z k] s' IH]. destruct n.
      + simpl in H0. exact H0.
      + simpl in H0. assert (Hnz : S n ≠ 0) by discriminate. destruct
(T_eq_dec x y) as [Heq | Hneq].
        × contradiction.
        × unfold InMultiset in H0.
          simpl in H0.
          destruct (T_eq_dec x y) as [Heq2 | Hneq2].
          contradiction.
          discriminate H0.
      + destruct (T_eq_dec x y) as [Heq | Hneq].
        × contradiction.
        × unfold InMultiset.
          simpl.
```

```
        destruct (T_eq_dec z y) as [Hzy | Hzy].
        reflexivity.
        simpl in H0.
        case n as [| Hn].
        −simpl in H0.
          unfold InMultiset in H0.
          simpl in H0.
          destruct (T_eq_dec z y) as [ Heq | _].
          ++contradiction.
          ++exact H0.
        −simpl in H0.
          destruct (T_eq_dec z x) as [ Hzx | Hzx].
          ++subst z.
            unfold InMultiset in H0.
            simpl in H0.
            destruct (T_eq_dec x y) as [Hcontr | _].
            contradiction.
            exact H0.
          ++unfold InMultiset in H0.
            simpl in H0.
            destruct (T_eq_dec z y) as [Hcontr | _].
            contradiction.
            exact (IH H0).
- intro. induction s as [| [z k] s' IH]. destruct n.
  + simpl. exact H0.
  + unfold InMultiset. unfold InMultiset in H0.
    simpl in H0.
    discriminate H0.
  + unfold InMultiset.
    unfold InMultiset in H0.
    simpl in H0.
    simpl. destruct n.
    × simpl.
      destruct (T_eq_dec z y) as [Hzy | Hzy].
      reflexivity.
      exact H0.
    × simpl.
      destruct (T_eq_dec z x) as [Hzx | Hzx].
      subst z.
      destruct (T_eq_dec x y) as [Hcontr | _].
      contradiction.
      −simpl.
        destruct (T_eq_dec x y) as [Hcontr | _].
        contradiction.
        exact H0.
      −simpl.
```

```
              destruct (T_eq_dec z y) as [Hzy | Hzy].
              reflexivity.
              assert (H' : InMultiset y s').
              unfold InMultiset.
              assumption.
              apply IH in H'.
              unfold InMultiset in H'.
              exact H'.
Qed.

Lemma prop_7_aux: ∀ x s,
  member x s = false → multiplicity x s = 0.
Proof.
  intros x s HnotMem.
  induction s.
  simpl.
  reflexivity.
  destruct a as [a an].
  simpl in HnotMem.
  simpl.
  destruct (T_eq_dec a x) as [Hax | Hax].
  discriminate HnotMem.
  exact (IHs HnotMem).
Qed.

Lemma prop_7 : ∀ x s, wf s → (multiplicity x s = 0 ↔ ¬InMultiset x s).
Proof.
  intros.
  split.
  - intro. unfold not, InMultiset. intro. induction s as [| [y n] s' IH].
    + simpl in H0, H1. discriminate.
    + simpl in H0, H1. destruct (T_eq_dec y x) as [Heq | Hneq].
      × destruct H as [Hn [Hh1 Hh2]]. lia.
      × apply IH. destruct H as [Hn [Hh1 Hh2]].
        – apply Hh2.
        – apply H0.
        – apply H1.
  - intro. induction s as [| [y n] s' IH].
    + simpl. reflexivity.
    + simpl. destruct (T_eq_dec y x) as [Heq | Hneq].
      × subst y.
        unfold InMultiset in H0.
        simpl in H0.
        destruct (T_eq_dec x x) as [_ | Hcontr].
        –contradiction.
        –contradiction.
      × unfold InMultiset in H0.
```

```
        simpl in H0.
        destruct (T_eq_dec y x) as [Hcontr | _].
        contradiction.
        simpl.
        apply Bool.not_true_is_false in H0.
        apply prop_7_aux.
        exact H0.
Qed.
```

Lemma prop_8 : $\forall$ x n s, multiplicity x (add x n s) = n + (multiplicity x s).

```
Proof.
  intros.
  induction s as [| [y k] s' IH].
  - simpl. destruct n.
    + simpl. reflexivity.
    + simpl. destruct (T_eq_dec x x) as [Heq | Hneq].
      × lia.
      × contradiction.
  - destruct n.
    + simpl. reflexivity.
    + simpl. destruct (T_eq_dec y x) as [Heq | Hneq].
      × subst y. simpl. destruct (T_eq_dec x x) as [Heq | Hneq].
        – lia.
        – contradiction.
      × simpl. destruct (T_eq_dec y x).
        – contradiction.
        – rewrite IH. reflexivity.
Qed.
```

Lemma prop_9 : $\forall$ x n y s, $x \neq y \rightarrow$ wf s $\rightarrow$ multiplicity y (add x n s) = multiplicity y s.

```
Proof.
  intros x n y s Hxy Hwf. revert n. revert x y Hxy. induction s as [| [z k]
s' IH]; intros.
  - simpl. destruct n.
    + reflexivity.
    + simpl.
      destruct (T_eq_dec x y) as [Heq | Hneq].
      × contradiction.
      × reflexivity.
  - simpl in Hwf. destruct Hwf as [Hk_pos [Hnot_in_s' Hwf_s']]. simpl.
destruct n as [| n'].
    + simpl. destruct (T_eq_dec z y) as [Heq1 | Hneq1].
      × subst. destruct (T_eq_dec y x) as [Heq2 | Hneq2].
        – symmetry in Heq2. contradiction.
        – simpl. reflexivity.
      × destruct (T_eq_dec z y).
```

```
      - contradiction.
      - reflexivity.
  + simpl. destruct (T_eq_dec z x) as [Heq | Hneq].
    × subst. destruct (T_eq_dec x y) as [Heq2 | Hneq2].
      - contradiction.
      - simpl. destruct (T_eq_dec x y) as [Heq | Hneq].
        ++ contradiction.
        ++ reflexivity.
    × destruct (T_eq_dec z y) as [Heq2 | Hneq2].
      - subst. simpl. destruct (T_eq_dec x y) as [Heq2 | Hneq2].
        ++ contradiction.
        ++ destruct (T_eq_dec y y) as [Heq3 | Hneq3].
          ** reflexivity.
          ** contradiction.
      - simpl. destruct (T_eq_dec z y) as [Heq3 | Hneq3].
        ++ contradiction.
        ++ exact ((IH Hwf_s') x y Hxy (S n')).
Qed.
```

2.2.4 Question 4

Propriétés pour removeOne

```
Lemma all_diff_means_not_in: ∀ a s,
  wf s →
  (∀ x occ, In (x, occ) s → x ≠ a) →
  (member a s = false).
Proof.
  intros a s Hwf H.
  induction s.
  simpl.
  reflexivity.
  destruct a0 as [y yn].
  simpl.
  destruct (T_eq_dec y a) as [Hya | Hya].
  - assert (Hcontr := H y yn).
    simpl in Hcontr.
    destruct Hcontr.
    left.
    reflexivity.
    exact Hya.
  - assert (H': (∀ (x : T) (occ : nat), In (x, occ) s → x ≠ a)).
    + intros x occ.
      assert (H' := H x occ).
      simpl in H'.
      intro HIn.
```

```
      apply H'.
      right.
      exact HIn.
    + destruct Hwf as [ _ [ _ Hwf]].
      exact (IHs Hwf H').
Qed.

Lemma multiplicity_removeOne_eq:
  ∀ x s, wf s → multiplicity x s > 0 →
  multiplicity x (removeOne x s) = multiplicity x s - 1.
Proof.
  intros x s Hwfs Hmul.
  induction s.
  simpl.
  reflexivity.
  destruct a as [a an].
  assert (Hwfcons := Hwfs).
  destruct Hwfs as [_ [_ Hwfs]].
  simpl.
  destruct (T_eq_dec a x) as [Hax | Hax].
  - subst x.
    destruct (an == 1) as [Han | Han].
    + rewrite Han.
      simpl.
      destruct Hwfcons as [H0 [H1 H2]].
      assert (H:= prop_7_aux a s).
      assert (member a s = false).
      × exact (all_diff_means_not_in a s H2 H1).
      × exact (H H3).
    + assert (Han' : an ≠ 1).
      assumption.
      apply Nat.eqb_neq in Han'.
      rewrite Han'.
      simpl.
      destruct (T_eq_dec a a) as [_ | Hcontr].
      × reflexivity.
      × contradiction.
  - simpl.
    simpl in Hmul.
    destruct (T_eq_dec a x) as [Hcontr | _].
    + contradiction.
    + exact (IHs Hwfs Hmul).
Qed.

Lemma multiplicity_removeOne_zero:
  ∀ x s, wf s → multiplicity x s = 0 →
  multiplicity x (removeOne x s) = 0.
```

```
Proof.
  intros x s Hwfs Hmul.
  induction s.
  simpl.
  reflexivity.
  destruct a as [a an].
  simpl.
  destruct (T_eq_dec a x) as [Hax | Hax].
  - subst x.
    destruct (an == 1) as [Han | Han].
    + rewrite Han.
      simpl.
      assert (an = 1).
      assumption.
      subst an.
      simpl in Hmul.
      destruct (T_eq_dec a a) as [_ | Hcontr].
      × discriminate Hmul.
      × contradiction.
    + assert (Han' : an ≠ 1).
      assumption.
      apply Nat.eqb_neq in Han'.
      rewrite Han'.
      simpl.
      simpl in Hmul.
      destruct (T_eq_dec a a) as [_ | Hcontr].
      × destruct Hwfs as [Hcontr [ _ _]].
        subst an.
        lia.
      × assumption.
  - simpl.
    simpl in Hmul.
    destruct (T_eq_dec a x) as [Hcontr | _].
    contradiction.
    destruct Hwfs as [_ [_ Hwfs]].
    exact (IHs Hwfs Hmul).
Qed.

Lemma multiplicity_removeOne_neq:
  ∀ x y s, wf s → x ≠ y →
  multiplicity y (removeOne x s) = multiplicity y s.
Proof.
  intros x y s Hwfs Hxy.
  induction s.
  simpl.
  reflexivity.
```

```
  destruct a as [a an].
  simpl.
  destruct (T_eq_dec a x) as [Hax | Hax].
  - subst x.
    destruct (an == 1) as [Han | Han].
    + rewrite Han.
      simpl.
      assert (an = 1).
      assumption.
      subst an.
      destruct (T_eq_dec a y).
      subst y.
      contradiction.
      reflexivity.
    + assert (Hd: an ≠ 1).
      assumption.
      assert(H' := Hd).
      apply Nat.eqb_neq in H'.
      rewrite H'.
      simpl.
      destruct (T_eq_dec a y).
      subst y.
      contradiction.
      reflexivity.
  - simpl.
    destruct (T_eq_dec a y).
    reflexivity.
    destruct Hwfs as [_ [ _ Hwfs]].
    exact (IHs Hwfs).
Qed.

Lemma InMultiset_removeOne_still_in:
  ∀ x s, wf s → multiplicity x s > 1 →
  InMultiset x (removeOne x s).
Proof.
  intros x s Hwfs Hmul.
  induction s.
  simpl in *. lia.
  unfold InMultiset in *.
  destruct a as [a an].
  simpl.
  destruct (T_eq_dec a x) as [Hax | Hax].
  - subst x.
    destruct (an == 1) as [Han | Han].
    + simpl.
      assert (an = 1).
```

```
      assumption.
      subst an.
      simpl in Hmul.
      destruct (T_eq_dec a a).
      lia.
      contradiction.
    + assert (Hd: an ≠ 1).
      assumption.
      assert(H' := Hd).
      apply Nat.eqb_neq in H'.
      rewrite H'.
      simpl.
      destruct (T_eq_dec a a).
      lia.
      contradiction.
  - simpl.
    simpl in Hmul.
    destruct (T_eq_dec a x).
    reflexivity.
    exact (IHs (proj2 (proj2 Hwfs)) Hmul).
Qed.

Lemma not_InMultiset_removeOne_gone:
  ∀ x s, wf s → multiplicity x s = 1 →
  ¬ InMultiset x (removeOne x s).
Proof.
  intros x s Hwfs Hmul.
  induction s.
  simpl in *.
  discriminate.
  destruct a as [a an].
  simpl.
  simpl in Hmul.
  destruct (T_eq_dec a x) as [Hax | Hax].
  - subst x.
    destruct (an == 1) as [Han | Han].
    + rewrite Han.
      simpl.
      assert (an = 1).
      assumption.
      subst an.
      assert (H' := Hwfs).
      destruct H' as [H0 [H1 H2]].
      assert (H' := all_diff_means_not_in a s H2 H1).
      unfold InMultiset.
      rewrite H'.
```

```
      discriminate.
    + subst an.
      assert (1 ≠ 1). assumption.
      contradiction.
  - unfold InMultiset.
    simpl.
    destruct (T_eq_dec a x).
    contradiction.
    destruct Hwfs as [_ [_ Hwfs]].
    assert (Hres := IHs Hwfs Hmul).
    unfold InMultiset in Hres.
    exact Hres.
Qed.

Lemma InMultiset_removeOne_other_1:
∀ x y s, wf s → x ≠ y →
  (InMultiset y (removeOne x s) → InMultiset y s).
Proof.
  intros x y s Hwfs Hxy H.
  induction s.
  simpl in *.
  assumption.
  unfold InMultiset.
  destruct a as [a an].
  simpl.
  destruct (T_eq_dec a y) as [Hay | Hay].
  reflexivity.
  simpl in H.
  destruct (T_eq_dec a x) as [Hax | Hax].
  - subst x.
    destruct (an == 1) as [Han | Han].
    + assert (an = 1). assumption.
      subst an.
      simpl in H.
      unfold InMultiset in H.
      exact H.
    + assert (H0 : an ≠ 1). assumption.
      apply Nat.eqb_neq in H0.
      rewrite H0 in H.
      unfold InMultiset in H.
      simpl in H.
      destruct (T_eq_dec a y).
      contradiction.
      exact H.
  - unfold InMultiset in H.
    simpl in H.
```

```
    destruct (T_eq_dec a y).
    contradiction.
    unfold InMultiset in IHs.
    destruct Hwfs as [_ [_ Hwfs]].
    exact (IHs Hwfs H).
Qed.

Lemma InMultiset_removeOne_other_2:
∀ x y s, wf s → x ≠ y →
  InMultiset y s → InMultiset y (removeOne x s).
Proof.
  intros x y s Hwfs Hxy H.
  induction s.
  simpl.
  assumption.
  unfold InMultiset in *.
  destruct a as [a an].
  simpl.
  destruct (T_eq_dec a x) as [Hax | Hax].
  - subst x.
    destruct (an == 1) as [Han | Han].
    + assert (an = 1). assumption.
      subst an.
      simpl.
      simpl in H.
      destruct (T_eq_dec a y) as [Hay | Hay].
      × contradiction.
      × exact H.
    + assert (H0 : an ≠ 1). assumption.
      apply Nat.eqb_neq in H0.
      rewrite H0.
      simpl.
      simpl in H.
      destruct (T_eq_dec a y) as [Hay | Hay].
      × contradiction.
      × exact H.
  - simpl.
    simpl in H.
    destruct (T_eq_dec a y).
    reflexivity.
    destruct Hwfs as [_ [_ Hwfs]].
    exact (IHs Hwfs H).
Qed.

Lemma InMultiset_removeOne_other:
  ∀ x y s, wf s → x ≠ y →
  (InMultiset y (removeOne x s) ↔ InMultiset y s).
```

```
Proof.
  intros x y s H Hxy.
  split.
  - intro H0. exact (InMultiset_removeOne_other_1 x y s H Hxy H0).
  - intro H0. exact (InMultiset_removeOne_other_2 x y s H Hxy H0).
Qed.
```

Propriétés pour removeAll

```
Lemma multiplicity_removeAll_eq:
  ∀ x s, wf s →
  multiplicity x (removeAll x s) = 0.
Proof.
  intros x s Hwfs.
  induction s.
  simpl.
  reflexivity.
  destruct a as [a an].
  simpl.
  destruct (T_eq_dec a x) as [Hax | Hax].
  - subst a.
    destruct Hwfs as [H0 [ H1 H2]].
    assert (H := all_diff_means_not_in x s H2 H1).
    exact (prop_7_aux x s H).
  - simpl.
    destruct (T_eq_dec a x).
    + contradiction.
    + destruct Hwfs as [_ [_ Hwfs]].
    exact (IHs Hwfs).
Qed.
```

```
Lemma multiplicity_removeAll_neq:
  ∀ x y s, wf s → x ≠ y →
  multiplicity y (removeAll x s) = multiplicity y s.
Proof.
  intros x y s Hwfs Hxy.
  induction s.
  simpl.
  reflexivity.
  destruct a as [a an].
  simpl.
  destruct (T_eq_dec a x) as [Hax | Hax].
  - destruct (T_eq_dec a y) as [Hay | Hay].
    subst a. contradiction.
    subst a.
    reflexivity.
```

```
  - simpl.
    destruct (T_eq_dec a y) as [Hay | Hay].
    subst a.
    reflexivity.
    destruct Hwfs as [_ [_ Hwfs]].
    exact (IHs Hwfs).
Qed.

Lemma not_InMultiset_removeAll:
  ∀ x s, wf s →
  ¬ InMultiset x (removeAll x s).
Proof.
  intros x s Hwfs.
  induction s.
  unfold InMultiset in *. discriminate.
  unfold InMultiset in *.
  destruct a as [a an].
  simpl.
  destruct Hwfs as [H0 [H1 H2]].
  destruct (T_eq_dec a x) as [Hax | Hax].
  - subst x.
    assert (H:= all_diff_means_not_in a s H2 H1).
    rewrite H.
    discriminate.
  - simpl.
    destruct (T_eq_dec a x).
    contradiction.
    exact (IHs H2).
Qed.

Lemma InMultiset_removeAll_other_1:
  ∀ x y s, wf s → x ≠ y →
  InMultiset y (removeAll x s) → InMultiset y s.
Proof.
  intros x y s Hwfs Hxy H.
  induction s.
  simpl in H. assumption.
  unfold InMultiset in *.
  destruct a as [a an].
  simpl.
  destruct Hwfs as [H0 [H1 H2]].
  destruct (T_eq_dec a x) as [Hax | Hax].
  - subst x.
    destruct (T_eq_dec a y) as [Hay | Hay].
    reflexivity.
    simpl in H.
    destruct (T_eq_dec a a).
```

```
    exact H.
    contradiction.
  - destruct (T_eq_dec a y) as [Hay | Hay].
    reflexivity.
    simpl in H.
    destruct (T_eq_dec a x).
    contradiction.
    simpl in H.
    destruct (T_eq_dec a y).
    contradiction.
    exact (IHs H2 H).
Qed.

Lemma InMultiset_removeAll_other_2:
  ∀ x y s, wf s → x ≠ y →
  InMultiset y s → InMultiset y (removeAll x s).
Proof.
  intros x y s Hwfs Hxy H.
  induction s.
  simpl in H. assumption.
  unfold InMultiset in *.
  destruct a as [a an].
  simpl.
  destruct Hwfs as [H0 [H1 H2]].
  destruct (T_eq_dec a x) as [Hax | Hax].
  - subst x.
    simpl in H.
    destruct (T_eq_dec a y) as [Hay | Hay].
    contradiction.
    assumption.
  - simpl.
    simpl in H.
    destruct (T_eq_dec a y) as [Hay | Hay].
    reflexivity.
    exact (IHs H2 H).
Qed.

Lemma InMultiset_removeAll_other:
  ∀ x y s, wf s → x ≠ y →
  (InMultiset y (removeAll x s) ↔ InMultiset y s).
Proof.
  intros x y s Hwfs Hxy.
  split.
  - intro H. exact (InMultiset_removeAll_other_1 x y s Hwfs Hxy H).
  - intro H. exact (InMultiset_removeAll_other_2 x y s Hwfs Hxy H).
Qed.

Lemma removeAll_not_member:
```

```
  ∀ x s, wf s → ¬ InMultiset x s →
  removeAll x s = s.
Proof.
  intros x s Hwfs HNotIn.
  induction s.
  simpl. reflexivity.
  destruct a as [a an].
  simpl.
  destruct (T_eq_dec a x) as [Hax | Hax].
  - subst x.
    unfold InMultiset in *.
    simpl in HNotIn.
    destruct (T_eq_dec a a).
    contradiction.
    contradiction.
  - f_equal.
    unfold InMultiset in *.
    simpl in HNotIn.
    destruct (T_eq_dec a x).
    contradiction.
    destruct Hwfs as [_ [_ Hwfs]].
    exact (IHs Hwfs HNotIn).
Qed.

Lemma removeAll_idempotent:
  ∀ x s, wf s →
  removeAll x (removeAll x s) = removeAll x s.
Proof.
  intros x s Hwfs.
  induction s.
  simpl. reflexivity.
  destruct a as [a an].
  simpl.
  destruct (T_eq_dec a x) as [Hax | Hax].
  - subst a.
    destruct Hwfs as [H0 [H1 H2]].
    assert (HnotIn: ¬InMultiset x s).
    + unfold InMultiset.
      assert (H:= all_diff_means_not_in x s H2 H1).
      rewrite H.
      discriminate.
    + exact (removeAll_not_member x s H2 HnotIn).
  - simpl.
    destruct (T_eq_dec a x).
    contradiction.
    f_equal.
```

```
    destruct Hwfs as [_ [_ Hwfs]].
    exact (IHs Hwfs).
Qed.

Lemma removeOne_not_member:
  ∀ x s, wf s → ¬ InMultiset x s →
  removeOne x s = s.
Proof.
  intros x s Hwfs HNotIn.
  induction s.
  simpl. reflexivity.
  simpl.
  destruct a as [a an].
  unfold InMultiset in *.
  simpl in HNotIn.
  destruct (T_eq_dec a x) as [Hax | Hax].
  contradiction.
  f_equal.
  destruct Hwfs as [_ [_ Hwfs]].
  exact (IHs Hwfs HNotIn).
Qed.
```

Chapter 3

Conclusion

À l'exception des questions **4.c** et **4.d** de **l'exercice 1**, tout a été prouvé.