Chapter 1

Library Pigeonhole

1.1 Pigeonhole Principle Solution

This is my solution of the pigeonhole principle exercise from Software Foundations. The solution has been separated from the rest of the Software Foundations dependencies so that it can be checked only using the Coq standard library. The solution was developed with Coq 8.6.

1.2 Helper Lemmas for In and AllIn

The Software Foundations exercise first suggests proving the following lemma. It may not be clear for a long time why this lemma is useful, but I could not prove *pigeonhole_principle* without it.

```
Lemma in_split : ∀ (X:Type) (x:X) (l:list X),
  In x l →
  ∃ l1 l2, l = l1 ++ x :: l2.
Proof.
  intros.
  induction l; [inversion H ||].
  destruct H as [H | H].
  ∃ nil; ∃ l.
  rewrite H; reflexivity.
  apply IHl in H.
  destruct H as [l1 [l2 H]].
  subst.
  ∃ (a::l1); ∃ l2.
  reflexivity.
Qed.
```

The following lemma helps in working with the In relation and list appends. It is clear that if $In\ x\ (a{+}{+}b)$ holds then either $In\ x\ a$ or $In\ x\ b$ holds.

```
Lemma in_app_split : ∀ T x (a b : list T),
  In x (a ++ b) ↔
  In x a ∨ In x b.
Proof.
  intros.
  split; intros H; induction a.
  - right; assumption.
  - destruct H as [H | H].
    left; left; assumption.
    apply IHa in H.
    destruct H as [H | H].
    left; right; assumption.
    right; assumption.
  - destruct H as [H | H].
    inversion H.
    assumption.
  - destruct H as [[H | H] | H].
    left; assumption.
    right; apply IHa.
    left; assumption.
    right; apply IHa.
    right; assumption.
Qed.
```

Software Foundations leaves it as an exercise to define a *repeats* property to be used in the *pigeonhole_principle* theorem.

The property should hold for a list that contains a repeated element.

There are several ways you can define *repeats* that allow for proving the pigeonhole principle. I have seen other solutions where *repeats* is defined using three constructors, however I use a single-constructor definition because it seemed most elegant.

```
Inductive repeats {X:Type} : list X → Prop :=
  | repeat_join : ∀ l1 l2, (∃ x , In x l1 ∧ In x l2) → repeats (l1 ++ l2).
```

The following lemma is very useful: it states that if you have a statement on the form *repeats* $(p{::}l)$ then it can be split into $In\ p\ l \lor repeats\ l$.

```
Lemma repeats_split : ∀ T (p:T) l,
  In p l ∨ repeats l ↔ repeats (p::l).
Proof.
  intros.
  split; intros H.
  - destruct H as [H | H].
    apply (repeat_join (p::nil)).
```

```
    ∃ p.
    split; [left; reflexivity | assumption].
    destruct H as [l1 l2 [x [Hl1 Hl2]]].
    apply (repeat_join (p::l1)).
    ∃ x.
    split; [right; assumption | assumption].
  - inversion H.
    destruct l1.
    destruct H1 as [x [H1l H1r]].
    inversion H1l.
    inversion H0.
    destruct H1 as [x [[H1l | H1l] H1r]]; [left; subst | right].
    + (*  left side *)
      apply in_app_split.
      right; assumption.
    + (*  right side *)
      apply repeat_join.
      ∃ x.
      split; assumption.
Qed.
```

The *allin* definition below is useful for making propositions more compact.

Other solutions that I have seen for the pigeonhole principle seem to use something resembling *allin* but which is defined as an inductive property with several constructors. I don't like that approach since it makes the *allin* construct needlessly powerful. The *allin* definition below is very weak - it does not add any new information about the property it describes.

```
Definition allin {T:Type} (l1 l2 : list T) : Prop :=
  ∀ u, In u l1 → In u l2.
```

Below are many lemmas about *In*, and *allin*. Most of them should be fairly obvious.

The next lemma is simply the fact that if an element u is in a list l, then u is in the concatenation p::l.

```
Lemma in_l_p : ∀ T (u p : T) l,
  In u l →
  In u (p::l).
Proof. intros; right; assumption. Qed.
```

A trivial fact: for all lists l, every element of the empty list is l.

```
Theorem nil_allin : ∀ T (l : list T), allin nil l.
Proof. intros × u []. Qed.
```

An *allin* proposition of the form *allin* (p::*l1*) *l2*, it implies *allin l1 l2*.

```
Lemma allin_pl_ll : ∀ T (p:T) l1 l2,
```

```
  allin (p::l1) l2 →
  allin l1 l2.
Proof.
  intros × H v Hv.
  apply (in_l_p _ v p) in Hv.
  apply H in Hv; assumption.
Qed.
```

The *allin* relation is transitive:

```
Lemma allin_trans : ∀ T (p:T) l1 l2,
  allin l1 l2 →
  In p l1 →
  In p l2.
Proof.
  intros T p [| t l1] l2 H Hpl1; inversion Hpl1;
    [subst t ||]; apply H; assumption.
Qed.
```

A proposition of the form *allin l1 l2* implies *allin* (*p*::*l1*) *l2* if *In p l2* holds.

```
Lemma allin_ll_pl : ∀ T (p:T) l1 l2,
  allin l1 l2 →
  In p l2 →
  allin (p::l1) l2.
Proof.
  intros × H Hp x Hx.
  destruct Hx as [Hx | Hx]; [subst; assumption ||].
  apply (allin_trans _ x l1) in H; assumption.
Qed.
```

A proposition of the form *allin l1 l2* implies *allin l1* (*p*::*l2*).

```
Lemma allin_ll_lp : ∀ T (p:T) l1 l2,
  allin l1 l2 →
  allin l1 (p::l2).
Proof.
  intros × H v Hv.
  apply H in Hv.
  apply in_l_p.
  assumption.
Qed.
```

A proposition of the form *allin* (*x*::*l1*) *l2* is equivalent to *In x l2* ∧ *allin l1 l2*.

```
Lemma allin_split : ∀ T (x:T) l1 l2,
  allin (x::l1) l2 ↔ In x l2 ∧ allin l1 l2.
Proof.
```

```
  intros T x l1 l2.
  split.
  - intros H; split.
    apply H.
    left; reflexivity.
    apply allin_pl_ll in H; assumption.
  - intros [H1 H2].
    apply allin_ll_pl; assumption.
Qed.
```

A proposition of the form *allin l1 (x::l2)* implies *In x l1* $\vee$ *allin l1 l2*.

```
Lemma allin_lp_info : ∀ T (x:T) l1 l2,
  allin l1 (x::l2) →
  In x l1 ∨ allin l1 l2.
Proof.
  intros.
  induction l1.
  - right; apply nil_allin.
  - apply allin_split in H.
    destruct H as [[Hx | Hx] H].
    + subst x.
      left; left; reflexivity.
    + apply IHl1 in H.
      destruct H as [H | H]; [left | right].
      right; assumption.
      apply allin_split; split; assumption.
Qed.
```

A proposition of the form *allin (x::l1) (y::l2)* implies $x = y \wedge$ (*In x l1* $\vee$ *allin l1 l2*) $\vee$ *In x l2* $\wedge$ *allin l1 (y::l2)*.

```
Lemma allin_pq_info : ∀ T (x y:T) l1 l2,
  allin (x::l1) (y::l2) →
  (x = y ∧ (In x l1 ∨ allin l1 l2)) ∨
  (In x l2 ∧ allin l1 (y::l2)).
Proof.
  intros.
  assert (allin (x::l1) (y::l2)) as G. { assumption. }
  apply allin_lp_info in H.
  destruct H as [[H | H] | H].
  - (*  H: x = y *)
    subst y.
    left; split; [reflexivity ||].
    apply allin_pl_ll in G.
```

```
    apply allin_lp_info in G.
    destruct G as [G | G]; [left | right]; assumption.
  - (*  H: In y l1 *)
    apply allin_split in G.
    destruct G as [[Gx | Gx] G]; [left | right]; split;
      try assumption; subst y; [reflexivity ||].
    apply allin_lp_info in G.
    assumption.
  - (*  H: allin (x::l1) l2 *)
    apply allin_split in H.
    destruct H as [Hx H].
    apply (allin_ll_lp _ y) in H.
    right; split; assumption.
Qed.
```

```
(*  Adding an element in the middle of a list increases the length by one: *)
```

```
Lemma len_mid_S : ∀ T (x : T) l r,
  length (l ++ (x :: r)) = S (length (l ++ r)).
Proof.
  intros.
  induction l; [reflexivity ||].
  simpl.
  rewrite IHl.
  reflexivity.
Qed.
```

This is a very specialized rewriting lemma used to rearrange a hypothesis inside *pigeonhole_principle*:

```
Lemma allin_rearrange1 : ∀ T (x z:T) l1 l2 r2,
  allin l1 (x :: (l2 ++ (z :: r2))) →
  allin l1 (z :: (x :: (l2 ++ r2))).
Proof.
  intros × H u Hu.
  apply H in Hu.
  destruct Hu as [Hu | Hu].
  - right; left; assumption.
  - apply in_app_split in Hu.
    destruct Hu as [Hu | [Hu | Hu]].
    + right; right; rewrite in_app_split.
      left; assumption.
    + left; assumption.
    + right; right; rewrite in_app_split.
```

```
    right; assumption.
Qed.
```

This is a very specialized rewriting lemma used to rearrange a hypothesis inside *pigeonhole_principle*:

```
Lemma allin_rearrange2 : ∀ T (z : T) l1 l2 r2,
  allin l1 (z :: (l2 ++ r2)) →
  allin l1 (l2 ++ (z::r2)).
Proof.
  intros x H u Hu.
  rewrite in_app_split.
  apply H in Hu.
  destruct Hu as [Hu | Hu].
  - right; left; assumption.
  - apply in_app_split in Hu.
    destruct Hu as [Hu | Hu]; [left | right]; [assumption ||].
    right; assumption.
Qed.
```

1.3 The Pigeonhole Principle

The pigeonhole principle is the fact that if you to place n marbles into m buckets, there most exist a bucket containing more than one marble if $n > m$.

The *pigeonhole_principle* theorem formalizes the pigeonhole principle by letting *l2* be a list of buckets, and *l1* a mapping of marbles to buckets.

Note that if two marbles are in the same bucket, then there is a repeated element in *l1*.

If the number of marbles is greater than the number of buckets, *length l1* > *length l2*, then there is a repeated element in *l1*.

```
Theorem pigeonhole_principle: ∀ (X:Type) (l1 l2:list X),
  (∀ x, In x l1 → In x l2) →
  length l2 < length l1 →
  repeats l1.
Proof.
  intros x H Hl.

  (* Rewrite the hypothesis as allin l1 l2: *)
  replace (∀ x, In x l1 → In x l2) with (allin l1 l2) in H; [| reflexivity].

  generalize dependent l2.

  induction l1 as [|x l1 IHl1]; intros l2 H Hl.
  inversion Hl.
  simpl in Hl.
  unfold lt in Hl.
```

```
(*  This rewrite is equivalent to applying the Sn_le_Sm__n_le_m
    theorem from Software Foundations: *)
rewrite ← Nat.succ_le_mono in Hl.

destruct l2 as [| y l2].
destruct (H x); left; reflexivity.

(*  H: allin (x::l1) (y::l2) *)
(*  Hl: length (y::l2) <= length l1 *)
(*  Make a copy of H as G. *)
assert (allin (x::l1) (y::l2)) as G. { assumption. }

(*  Rewrite H to useful conclusions. *)
apply (allin_pq_info) in H.
destruct H as [[Hx [H | H]] | [Hx Hy]].

- (*  H: x = y /\ In x l1 *)
  apply repeats_split; left; assumption.

- (*  H: x = y /\ allin l1 l2 *)
  apply IHl1 in H; [| assumption].
  apply repeats_split; right; assumption.

- (*  H: In x l2 /\ allin l1 (y::l2) *)
  apply allin_lp_info in Hy.
  destruct Hy as [Hy | Hy].

  + (*  Hy: y is in l1 *)
    apply in_split in Hx.
    destruct Hx as [xl [xr Hx]].
    apply in_split in Hy.
    destruct Hy as [yl [yr Hy]].

    (*  Rearrange the G hypothesis into J: *)
    assert (In x (l1) ∨ allin (l1) (xl ++ (y::xr))) as J. {
      subst l1 l2.
      apply allin_pl_ll in G.
      apply allin_rearrange1 in G.
      apply allin_lp_info in G.
      destruct G as [G | G]; [left | right]; [assumption ||].
      apply allin_rearrange2 in G.
      assumption. }
    destruct J as [J | J].

    (*  J: x is in l1 *)
    apply repeats_split; left; assumption.

    (*  J: allin l1 (x1 ++ y::xr) *)
    apply repeats_split; right.
```

```
    apply (IHl1 (xl ++ y::xr)).
    assumption.

    (*  Prove the length condition of induction hypothesis. *)
    simpl in Hl.
    subst l1 l2.
    repeat (rewrite len_mid_S).
    repeat (rewrite len_mid_S in Hl).
    assumption.

  + (*  Hy: all of l1 are in l2 *)
    apply IHl1 in Hy.
    apply repeats_split; right; assumption.
    assumption.
Qed.
```

□

Jesper Öqvist, 2017 − 03 − 17