CS 398 Final Report

Author: Mohammed Aaqel Shaik Abdul Mazeed (mshai4@uic.edu)
UIN: 652330487
Major: Computer Science
Term: Spring 2022
CRN: 10725
Credit Hours: 3
Instructor: William Mansky (mansky1@uic.edu)

Aim: To learn to use the Coq proof assistant and use it to verify programs of interest.

Result:
I read through *Software Foundations Volume 1: Logical Foundations* for the first 8 or so weeks of class and this served as a good introduction point to the field of automated proof assistants. I was able to form a strong basis for how the Coq proof assistant worked, and what it was used for.

For the next 4 weeks, Professor Mansky suggested a more hands-on approach, where we worked through examples to learn about the different tactics used in Coq and their various uses.

Finally, I expressed interest in using Coq to work on the MU Problem from *Gödel, Escher, Bach: an Eternal Golden Braid.* I aimed to use the proof assistant to show why the problem was unsolvable. For the last 4 weeks, I worked on this, and Professor Mansky guided me through the different facets of problem-solving in Coq.

Attached is the final version of the file we worked on for the MU problem:

```
(** Name:       mu.v
    Goal:       Prove that every string generated from applying the
                rules of the MU Puzzle will have at least one 'I'.
    Author:     Mohammed Aaqel Shaik Abdul Mazeed (mshai4@uic.edu)
    Instructor: William Mansky                    (mansky1@uic.edu)
**)

Require Import List.
Import ListNotations.
Require Import Arith.
Require Import Lia.

(** Declare characters **)
```

```
Inductive char : Type :=
  | M
  | I
  | U.

(** Model rules of the puzzle **)
Inductive MIU : list char -> Prop :=
(* 0. Base string *)
  | base : MIU [M;I]
(* 1. For any string xI, we can produce a string xIU.
      That is, we append 'U' to the string. *)
  | append : forall x, MIU (x ++ [I]) -> MIU (x ++ [I;U])
(* 2. For any string Mx, we can produce a string Mxx.
      That is, we double the string after M. *)
  | double : forall x, MIU ([M] ++ x) -> MIU ([M] ++ x ++ x)
(* 3. For any string xIIIy, we can produce a string xUy.
      That is, we replace 'III' with 'U' in the string. *)
  | replace : forall x y, MIU (x ++ [I;I;I] ++ y) -> MIU (x ++ [U]
++ y)
(* 4. For any string xUUy, we can produce a string xy.
      That is, we remove 'UU' from the string. *)
  | remove : forall x y, MIU (x ++ [U;U] ++ y) -> MIU (x ++ y).

(** Proofs **)

(* Prove that MIU is a valid string derived from the rules
established.
   Here, we start with MIU and work our way backwards to get the
base string of MI.
 *)
Lemma miu: MIU [M;I;U].
Proof.
apply append with (x := [M]).
apply base.
Qed.

(* Prove the same as above, only this time, moving forward.*)
Lemma miu_forward: MIU [M;I;U].
Proof.
pose proof base.
apply append with (x:= [M]) in H.
apply H.
Qed.
```

```
(* Prove that we can derive MU from MUU *)
Lemma muu_mu: MIU [M;U;U] -> MIU [M;U].
Proof.
intros. remember [M;U;U] as s. induction H; try discriminate.
- (*1*)
  Search (_ ++ _ = _ -> _).
  pose proof (app_inv_tail ([U]) (x ++ [I]) ([M;U])) as Hinvtail.
  rewrite<- app_assoc in Hinvtail.
  apply Hinvtail in Heqs.
  apply app_inj_tail with (y := [M]) in Heqs.
  destruct Heqs. discriminate.
- (*2*)
  destruct x; inversion Heqs.
  subst.
  destruct x; inversion H2.
  + apply H.
  + Search app nil eq. apply app_eq_nil in H3.
  destruct H3 as [? ?]. discriminate.
Abort. (* This lemma is most likely unprovable*)

(* Prove that for any 2 char x and y, x = y or x != y *)
Lemma eq_dec : forall x y : char, {x = y} + {x <> y}.
Proof.
  decide equality.
Defined.

(*Experimentation
Theorem easy : forall p q:Prop, (p->q)->(~q->~p).
Proof. intros. intro. apply H0. apply H. exact H1. Qed.

Lemma contrapositive : forall P Q, ~Q -> ~P.
Proof.
  intros P Q HQ HP. contradiction HQ. Abort.

Theorem plus_comm_test: forall n m p: nat,
  m + p + (n + p) = m + n + 2 * p.
Proof. intros. rewrite plus_assoc. simpl. rewrite <- plus_n_O.
Abort.
*)

Lemma plus2mult: forall (n: nat), n + n = 2 * n.
Proof. intros. simpl. rewrite <- plus_n_O. reflexivity. Qed.

(* Prove that n mod 3 <> 0 -> (n + n) mod 3 = 0 *)
```

```
Lemma mod_helper: forall (n : nat), n mod 3 <> 0 -> (n + n) mod 3
<> 0.
intros. intro Hmod. contradiction H.
rewrite plus2mult in Hmod.
rewrite Nat.mod_divide in Hmod; try discriminate.
rewrite Nat.mod_divide; try discriminate.
apply Nat.gauss in Hmod; try reflexivity. exact Hmod.
Qed.

(* This makes it so that mod isn't expanded when simpl is called *)
Locate modulo.
Arguments Nat.modulo (!_) _.


(* Prove that the number of I's in a list will never be a multiple
of 3 *)
Lemma never3: forall l, MIU l -> count_occ (eq_dec) (l : list char)
I mod 3 <> 0.
Proof.
intros. induction H.
- (*0*)
  simpl. discriminate.
- (*1*)
  rewrite count_occ_app in IHMIU.
  rewrite count_occ_app.
  simpl. simpl in IHMIU.
  apply IHMIU.
- (*2*)
  rewrite count_occ_app in IHMIU.
  rewrite count_occ_app.
  rewrite count_occ_app.
  simpl. simpl in IHMIU.
  apply mod_helper. apply IHMIU.
- (*3*)
  rewrite count_occ_app.
  rewrite count_occ_app.
  simpl in IHMIU.
  rewrite count_occ_elt_eq in IHMIU; try reflexivity.
  rewrite count_occ_elt_eq in IHMIU; try reflexivity.
  rewrite count_occ_elt_eq in IHMIU; try reflexivity.
  replace (S (S (S (count_occ eq_dec (x ++ y) I))))
  with (count_occ eq_dec (x ++ y) I + (1 * 3)) in IHMIU by lia.
  rewrite Nat.mod_add in IHMIU; try discriminate.
  rewrite count_occ_app in IHMIU.
```

```
  simpl. apply IHMIU.
- (*4*)
  simpl in IHMIU.
  rewrite count_occ_elt_neq in IHMIU; try discriminate.
  rewrite count_occ_elt_neq in IHMIU; try discriminate.
  apply IHMIU.
Qed.

Theorem neverMU: ~MIU [M; U].
Proof.
intro. apply never3 in H. apply H. contradiction.
Qed.
```

Using the helper lemmas we proved above, we were able to prove the final theorem that we would never obtain a string ‘MU’ with the rules we’ve established.