CIS 5000 Midterm 8 October 2024

SOLUTIONS

1. **True/False Miscellany** (10 points)

   1 point each unless otherwise specified.

   (a) In Coq, there are *no* syntactic terms `X`, `Y`, and `Z`, such that `X : Y` and `Y : Z` are valid type assignments.
   ☐ True ☒ False

   (b) Recall that *functional extensionality* is the principle that `∀ (f g : X → Y), (∀(x:X), f x = g x) → f = g`. Functional extensionality can be added as a *consistent axiom* to Coq's base logic.
   ☒ True ☐ False

   (c) To ensure consistency, every *tactic* used to interactively construct a Coq proof term must terminate.
   ☐ True ☒ False

   (d) In Coq, every function of type `nat → bool` represents a *decidable proposition* about the natural numbers.
   ☒ True ☐ False

   (e) Because Coq is a *constructive logic* there does not exist a proposition `P` such that `P ∨ ∼P` is provable.
   ☐ True ☒ False

   (f) (2 points) Which of the following are *dependent types*? (choose all that apply)
   ☐ `nat → Prop`
   ☐ `nat → bool`
   ☐ `fun (x:nat) ⇒ x = 3`
   ☒ `∀ (x:nat), x = 3`
   ☐ `∀ (x:nat), bool`

   Consider the following function definition that calculates n^m by repeated multiplication:

```
Fixpoint exp (n:nat) (m:nat) : nat :=
  match m with
  | 0 ⇒ 1
  | S m' ⇒ n * (exp n m')
  end.
```

   (g) We can replace the keyword **`Fixpoint`** in the above with **`Definition`** without changing the meaning of the program.
   ☐ True ☒ False

   (h) (2 points) Suppose we want to prove the following theorem:

```
Lemma exp_mult : ∀ n m1 m2,
    (exp n m1) * (exp n m2) = (exp n (m1 + m2)).
```

   The proof will be smoothest if, after we do `intros n m1 m2`, we continue by doing: (choose one)
   ☐ `induction n`
   ☒ `induction m1`
   ☐ `destruct m2`
   ☐ `destruct (m1 + m2)`

2. **Types and Terms** (16 points)

The following questions refer to this inductively defined proposition:

```
Inductive le : nat → nat → Prop :=
| le_0 (n : nat)                    : le 0 n
| le_S_both (n m : nat) (H: le n m) : le (S n) (S m).
```

For each of the following terms, write its type, or write "ill typed" if that it the case. (2 points each)

(a) `S (S 0)`

: `nat`

(b) `le_0`

: `∀ n. le 0 n`

(c) `(le_S_both 4)`

: `∀ m : nat, le 4 m → le 5 (S m)`

(d) `∀ n, le (S n) n`

: `Prop`

(e) `le 2 1`

: `Prop`

(f) `fun (n:nat) ⇒ le n (S n)`

: `nat → Prop`

(f) (2 points) Does there exist a closed term of type `(le 1 2)`? If so, give one.

☐ no such term exists ☒ yes: `le_S_both 0 1 (le_0 1)`

(g) (2 points) Does there exist closed a term of type `(le 2 1)`? If so, give one.

☒ no such term exists ☐ yes:

3. **Logic and Tactics** (18 points)

Consider the following four logical lemmas and the five proof attempts below. Match each lemma to its corresponding proof. Note that Proof E ends in `Abort`, which indicates that the lemma is *not provable*. Each of the proofs is used *at most once*.

(a) (3 points)

```
Lemma logic1 : ∀ (P Q R: Prop), ((P ∧ Q) → R) → Q → (P → R).
```

Proved by: ☐ **A** ☐ **B** ☐ **C** ☒ **D** ☐ **E** (unprovable)

(b) (3 points)

```
Lemma logic2 : ∀ (P Q R: Prop), ((P ∧ Q) → R) → ∼R → (P → ∼Q).
```

Proved by: ☐ **A** ☐ **B** ☒ **C** ☐ **D** ☐ **E** (unprovable)

(c) (3 points)

```
Lemma logic3 : ∀ (P Q R: Prop), ((P ∧ Q) → R) → ((P → R) ∨ (Q → R)).
```

Proved by: ☐ **A** ☐ **B** ☐ **C** ☐ **D** ☒ **E** (unprovable)

(d) (3 points)

```
Lemma logic4 : ∀ (P Q R: Prop), ((P ∨ Q) → R) → ((P → R) ∨ (Q → R)).
```

Proved by: ☒ **A** ☐ **B** ☐ **C** ☐ **D** ☐ **E** (unprovable)

Proof A

```
Proof.
  intros P Q R H1.
  left. intros H2. apply H1. left. apply H2.
Qed.
```

Proof B

```
Proof.
  intros P Q R [H1 | H1] H2.
  - left. split. apply H1. apply H2.
  - right. split. apply H1. apply H2.
Qed.
```

Proof C

```
Proof.
  intros P Q R H1 H2 H3 H4.
  apply H2. apply H1. split. apply H3. apply H4.
Qed.
```

Proof D

```
Proof.
  intros P Q R H1 H2 H3.
  apply H1. split. apply H3. apply H2.
Qed.
```

Proof E

```
Proof.
  intros P Q R H1.
Abort.
```

Each question below refers to the state of the proof shown below (i.e., they are independent questions—not a suggested sequence of operations).

```
A, B, C : Prop
a : A
H : A -> (B = C)
---------------
B = C
```

(e) (2 points) In the proof state above, using the `reflexivity` tactic would: (choose one)

☐ Succeed completely, finishing the proof.

☐ Succeed leaving 1 remaining goal.

☐ Succeed leaving 2 remaining goals.

☒ Fail, with a unification error `Unable to unify "C" with "B"`

(f) (2 points) In the proof state above, using the `apply H` tactic would: (choose one)

☐ Succeed completely, finishing the proof.

☒ Succeed leaving 1 remaining goal.

☐ Succeed leaving 2 remaining goals.

☐ Fail, with a unification error `Unable to unify "C" with "B"`

(g) (2 points) In the proof state above, using the `rewrite H` tactic would: (choose one)

☐ Succeed completely, finishing the proof.

☐ Succeed leaving 1 remaining goal.

☒ Succeed leaving 2 remaining goals.

☐ Fail, with a unification error `Unable to unify "C" with "B"`

4. **Functional Programming and Proofs about Functions** (20 points total)

A *rope* is a polymorphic tree-based data structure that stores a sequence of elements like a *list* but that also supports a constant-time "append" operation. The definition is given below, where the `Inc` constructor "includes" a list into a rope, and the `App` constructor appends two ropes (in constant time).

```
Inductive rope (X:Type) : Type :=
| Inc (l : list X)
| App (r1 : rope X) (r2 : rope X).

Arguments Inc {X} _.
Arguments App {X} _ _.

(* One rope representation of the list [1;2;3;4;5] *)
Example rope1 : rope nat := App (Inc [1;2;3]) (Inc [4;5]).
```

(a) (5 points) Each `rope` can be converted to a `list` by appending the lists found at the `Inc` leaves in order from left to right. For example, we should have: `rope_to_list rope1 = [1;2;3;4;5]`. Complete the definition of such a function, `rope_to_list`, below. It should use the list `app` function (a.k.a. `++`), which is defined in the Appendix.

```
Fixpoint rope_to_list {X} (r : rope X) : list X :=
  match r with
  | Inc x ⇒ x
  | App r1 r2 ⇒ (rope_to_list r1) ++ (rope_to_list r2)
  end.
```

(b) (4 points) Note that the representation of "the empty sequence" as a rope is *not* unique. Give two distinct expressions `r1` and `r2` of type `rope nat` such that `(rope_to_list r1) = [] = (rope_to_list r2)` but `r1 <> r2`.

```
Example r1 : rope nat := Inc [].
Compute (rope_to_list r1). (* ≡⇒ [] *)

Example r2 : rope nat := App (Inc []) (Inc []).
Compute (rope_to_list r2). (* ≡⇒ [] *)
```

(c) (3 points) Complete the definition of the following function that "appends" two lists to produce a rope. (Note that it is not recursive!):

```
Definition list_app_rope {X} (l1 l2 : list X) : rope X :=

  App (Inc l1) (Inc l2).
```

Your implementations above should satisfy the following correctness lemma:

```
Lemma list_app_rope_correct : ∀ {X} (l1 l2 : list X),
    rope_to_list (list_app_rope l1 l2) = l1 ++ l2.
Proof.
  reflexivity.
Qed.
```

(d) Looking up the n^{th} element of a rope is a bit more involved than doing so for a list. First, we need a helper function that can compute the length of a rope:

```
Fixpoint rope_length {X} (r : rope X) : nat := (* ... omitted *)
```

We use it to define this lookup function:[1]

```
Fixpoint rope_nth_error {X} (r : rope X) (n:nat) : option X :=
  match r with
  | Inc l ⇒ nth_error l n
  | App r1 r2 ⇒
      if n <? rope_length r1 then rope_nth_error r1 n
      else rope_nth_error r2 (n - rope_length r1)
  end.
```

The correctness of this function can be stated as follows. As above, it relates the rope version to the list version of `nth_error`. The definition of `nth_error` and some related lemmas are stated in the Appendix.

```
Lemma rope_nth_error_correct : ∀ {X} (r:rope X) (n:nat),
    rope_nth_error r n = nth_error (rope_to_list r) n.
Proof.
  intros X r.
  induction r as [l | r1 IHr1 r2 IHr2].
  (* details omitted *)

Qed.
```

The following questions probe your understanding of how the omitted proof above would be structured. You do not have to write out the whole proof, just answer these questions!

i. (2 points) At the point where the details are omitted above, how many goals remain in this proof? (choose one)

☐ 1 ☒ 2 ☐ 3 ☐ 4 ☐ 5

ii. (2 points) Which of the following steps can be useful in completing this proof? (choose *all* that apply)

☐ `destruct (rope_length r1)`

☒ `destruct (n <? rope_length r1) eqn:HLT`

☐ `destruct (n -rope_length r1) eqn:HSUB`

☒ `destruct (Reflect (n < rope_length r1) (n <? rope_length r1))`

☐ `destruct ((rope_length r1) -(rope_length r2)) eqn:HSUB`

iii. (2 points) Since `rope_nth_error` relies on the helper function `rope_length`, we will need a lemma about it too. Which one is needed for the proof above? (choose one)

☐ `Lemma rope_length_cons : ∀X (r : rope X) (x:X), S (rope_length r) = length (x :: (rope_to_list r)).`

☐ `Lemma rope_length_app : ∀X (r1 r2 : rope X), rope_length (App r1 r2) = (rope_length r1) + (rope_length r2).`

☒ `Lemma rope_length_correct : ∀X (r : rope X), rope_length r = length (rope_to_list r).`

☐ `Lemma rope_length_Inc : ∀X (l:list X), rope_length (Inc l) = length l.`

iv. (2 points) Which one of the lemmas from the appendix will be needed in the same case of the proof as `IHr1`? (choose one)

☐ `app_length`

☒ `nth_error_app1`

☐ `nth_error_app2`

☐ `nat_nlt_ge`

[1] This function calculates the length of the rope `r1` twice, which is inefficient. A better implementation would cache that result, but we ignore that here for the sake of simplicity.

5. **Defining Inductive Propositions** (10 points total)

This problem asks you to define inductive propositions that work with the same rope type as defined in the previous problem. We repeat the definition here for your convenience.

```
Inductive rope (X:Type) : Type :=
| Inc (l : list X)
| App (r1 : rope X) (r2 : rope X).

Arguments Inc {X} _.
Arguments App {X} _ _.

(* One rope representation of the list [1;2;3;4;5] *)
Example rope1 : rope nat := App (Inc [1;2;3]) (Inc [4;5]).
```

(a) (2 points) Consider the following inductively defined predicate:

```
Inductive rope_pred {X} : rope X → Prop :=
| p_Inc : rope_pred (Inc [])
| p_App : ∀ r1 r2,
    rope_pred r1 →
    rope_pred r2 →
    rope_pred (App r1 r2).
```

Which of the following informal descriptions best describes the property that holds of the rope `r` when we have evidence for `rope_pred r`? (choose one)

☐ This predicate holds when `r` contains at least one occurrence of the `Inc` constructor applied to `[]`.

☒ This predicate holds when *every* occurrence of the `Inc` constructor in `r` is applied to `[]`.

☐ This predicate holds when the *first* occurrence of an `Inc` constructor in `r` is applied to `[]`.

☐ This predicate holds when the *last* occurrence of an `Inc` constructor in `r` is applied to `[]`.

(b) (8 points) We might want to define the set of *all* ropes that correspond to a given list `l`. For example, we know that the example `rope1 = App (Inc [1;2;3]) (Inc [4;5])` corresponds to the list `[1;2;3;4;5]`, but so does `rope2 = Inc [1;2;3;4;5]` and also `rope3 = App (Inc [1;2]) (Inc [3;4;5])`. In the space below, complete the inductively defined proposition `list_rope_spec` that defines this relation.

Note: your definition should *not* use `rope_to_list`. Instead, we want to prove a separate correctness theorem that connects them—see `list_rope_spec_correct` given in the appendix. (The exact order of the named hypotheses shown there might differ from those that would be needed for your definition.)

```
Inductive list_rope_spec {X} : list X → rope X → Prop :=

| ltr_Inc : ∀ l, list_rope_spec l (Inc l)

| ltr_App : ∀ l1 l2 r1 r2,
    list_rope_spec l1 r1 →
    list_rope_spec l2 r2 →
    list_rope_spec (l1 ++ l2) (App r1 r2).
```

6. **Working with Inductive Propositions** (16 points)

Recall the `ev` predicate (see the Appendix), which characterizes the set of *even* natural numbers.

(a) For each lemma below, indicate what tactics, besides `intros` and `apply` are needed to finish the proof. (Or mark it "not provable" if that is the case.)

i.

```
Lemma ev_2_plus_n : ∀ n, ev n → ev (S (S n)).
```

- ☐ Not provable.
- ☒ Provable without using `inversion` or `induction`.
- ☐ Provable using only `inversion` (but not `induction`).
- ☐ Provable only by using `induction`.

ii.

```
Lemma ev_SS_n : ∀ n, (ev (S (S n))) → (ev n).
```

- ☐ Not provable.
- ☐ Provable without using `inversion` or `induction`.
- ☒ Provable using only `inversion` (but not `induction`).
- ☐ Provable only by using `induction`.

iii.

```
Lemma ev_not_S: ∀ n, ev n → ~ ev (S n).
```

- ☐ Not provable.
- ☐ Provable without using `inversion` or `induction`.
- ☐ Provable using only `inversion` (but not `induction`).
- ☒ Provable only by using `induction`.

iv.

```
Lemma ev_n_1 : ∀ n, ev n → ev 1.
```

- ☒ Not provable.
- ☐ Provable without using `inversion` or `induction`.
- ☐ Provable using only `inversion` (but not `induction`).
- ☐ Provable only by using `induction`.

v.

```
Lemma not_ev_5 : ~ (ev 5).
```

- ☐ Not provable.
- ☐ Provable without using `inversion` or `induction`.
- ☒ Provable using only `inversion` (but not `induction`).
- ☐ Provable only by using `induction`.

(b) (3 points) Consider the following proof (taken from the HW) that does induction on the evidence `ev n`.

```
Lemma ev_plus : ∀ n m, ev n → ev m → ev (n + m).
Proof.
  intros n m Hn Hm.
  induction Hn.
  - apply Hm.
  - (* HERE *) simpl. apply ev_SS. apply IHHn.
Qed.
```

The proof state at the point marked `(* HERE *)` in the above script is:

```
m, n : nat
Hn : ev n
Hm : ev m
IHHn : ev (n + m)
----------------------------------
ev (S (S n) + m)
```

Now consider the following variation that uses `generalize dependent` at the point marked `CHANGE!`

```
Lemma ev_plus' : ∀ n m, ev n → ev m → ev (n + m).
Proof.
  intros n m Hn Hm.
  generalize dependent m.  (* ←--- CHANGE! *)
  induction Hn.
  - intros m Hm. apply Hm.
  - (* HERE *) intros m Hm. simpl. apply ev_SS. apply IHHn. apply Hm.
Qed.
```

Which of the following will be the type of `IHHn` at the point marked `(* HERE *)` for this new variant of the proof? (choose one)

☐ `IHHn : ∀m, ev (n + m)`

☐ `IHHn : ∀n m, ev (n + m)`

☒ `IHHn : ∀m, ev m → ev (n + m)`

☐ `IHHn : ∀m, ev n → ev m → ev (n + m)`

(c) (3 points) Following on from part (b): If instead, we try to do induction on `n` rather than `Hn`, we end up with:

```
Lemma ev_plus'' : ∀ n m, ev n → ev m → ev (n + m).
Proof.
  intros n.
  induction n. (* ←--- CHANGE! Induction on n *)
  - intros m Hn Hm. apply Hm.
  - (* HERE *) intros m Hn Hm.
Abort.
```

With this proof state:

```
n : nat
IHn : forall m : nat, ev n -> ev m -> ev (n + m)
m : nat
Hn : ev (S n)
Hm : ev m
--------------------------------------
ev (S n + m)
```

But from here, we will not be able to complete the proof. Which of the following is the best explanation why? (choose one)

☐ Using the tactic `inversion Hn` will fail, thereby preventing us from learning more information about `(S n)`.

☐ The only possibly applicable constructor is `ev_SS`, but it can never be used in this case, no matter what sequence of tactics we try to use.

☒ We have `Hn : ev (S n)`, but that means that `(S n)` is even, so `n` must be odd and there is no way to invoke the induction hypothesis, which requires evidence for `ev n`.

☐ We have reached a contradictory state—it is not possible for `ev (S n)`, `ev m` and `ev (S n + m)` to be provable at the same time.

CIS 5000 2024 Midterm 1 Appendices

(Do not write answers in the appendices. They will not be graded)

Coq Library Definitions

```
(* OPTIONS  *)

Inductive option (X:Type) : Type :=
  | Some (x : X)
  | None.

Arguments Some {X} _.
Arguments None {X}.

(* LISTS *)

(* appends two lists *)
Fixpoint app {X : Type} (l1 l2 : list X) : list X :=
  match l1 with
  | nil      ⇒ l2
  | cons h t ⇒ cons h (app t l2)
  end.

Notation "x ++ y" := (app x y)
                     (at level 60, right associativity).

(* computes the length of a list *)
Fixpoint length {X : Type} (l : list X) : nat :=
  match l with
  | nil ⇒ 0
  | cons _ l' ⇒ S (length l')
  end.

Fixpoint nth_error {X : Type} (l : list X) (n : nat)
                   : option X :=
  match l with
  | nil ⇒ None
  | a :: l' ⇒ match n with
               | O ⇒ Some a
               | S n' ⇒ nth_error l' n'
               end
  end.
```

Coq Lemmas

```
Lemma app_length : ∀ {A : Type} (l l' : list A),
          length (l ++ l') = length l + length l'.

Lemma nth_error_app1 : ∀ (A : Type) (l l' : list A) (n : nat),
    n < length l →
    nth_error (l ++ l') n = nth_error l n.

Lemma nth_error_app2 : ∀ (A : Type) (l l' : list A) (n : nat),
          length l ≤ n →
          nth_error (l ++ l') n = nth_error l' (n - length l).

Lemma nat_nlt_ge : ∀ n m : nat, ∼ n < m → m ≤ n.
```

List–Rope Correspondence

```
Lemma list_rope_spec_correct : ∀ {X} (r : rope X) (l : list X),
    list_rope_spec l r →
    rope_to_list r = l.
Proof.
  intros X r l H.
  induction H as [l | l1 l2 r1 r2 H1 IHr1 H2 IHr2].
  - reflexivity.
  - simpl. rewrite IHr1. rewrite IHr2.
    reflexivity.
Qed.
```

ev Predicate

Characterizes the set of even natural numbers.

```
Inductive ev : nat → Prop :=
| ev_0                    : ev 0
| ev_SS (n : nat) (H : ev n) : ev (S (S n)).
```