CIS 5000 Midterm 11 October 2022

SOLUTIONS

1. **Typing 1** (9 points) ($1\frac{1}{2}$ points each)

   What is the type of each of the following Coq expressions? If it has no type, write "ill-typed."

   (a) `true`
   `bool`

   (b) `False`
   `Prop`

   (c) `4 = 4`
   `Prop`

   (d) `fun (X:Prop) => X -> X`
   `Prop -> Prop`

   (e) `fun (X:Type) (f: X -> (X -> X)) (x:X) => f x`
   `forall X : Type, (X -> X -> X) -> X -> X -> X`

   (f) `forall (X:Prop), X \/ (X -> False)`
   `Prop`

2. **Typing 2** (9 points) ($1\frac{1}{2}$ points each)

   Write a Coq expression for each of the following types. Write "empty" if there are no expressions of that type.

   (a) `Prop`

   `True`

   (b) `nat -> Prop`

   `fun x => x = x`

   (c) `101 <= 100`

   empty

   (d) `forall (X : Type), X -> (X -> X)`

   `fun (X:Type) (x:X) (y:X) => x`

   (e) `(bool -> bool) -> bool`

   `fun (f : bool -> bool) => f true`

   (f) `forall (X Y : Type), X -> Y`

   empty

3. **Tactics!** (18 points) (3 points each)

Fill in the partially completed proof states such that the requirement(s) are satisfied.

Note: Many solutions are possible.

(a) Proof state:

```
m, n: nat
H: m = 1 \/ n = 2
----------------------------------
(* IGNORE THIS LINE *)
```

Requirement:
(1) `destruct H.` will generate **two** subgoals

(b) Proof state:

```
m, n: nat
H: m = 1 /\ n = 2
----------------------------------
(* IGNORE THIS LINE *)
```

Requirement:
(1) `destruct H.` will generate **one** subgoal

(c) Proof state:

```
H: False
----------------------------------
(* IGNORE THIS LINE *)
```

Requirement:
(1) `destruct H.` will generate **zero** subgoals

(d) Proof state:

```
m, n: nat
H: S m = S n
----------------------------------
m = n
```

Requirements:
(1) `injection H as Hmn. apply Hmn.` will solve the goal
(2) `f_equal. apply H.` will not

(e) Proof state:

```
n: nat
H: n = 3
----------------------------------
n + 1 = 4
```

Requirements:
(1) `rewrite H. reflexivity.` will solve the goal
(2) `apply H.` will not

(f) Proof state:

```
P: Prop
Q: Prop
H: P /\ Q
----------------------------------
P /\ Q
```

Requirements:
(1) `apply H.` will solve the goal
(2) `rewrite H. reflexivity.` will not

4. **[Standard Track Only] Functional Programming** (20 points total)

We define a binary tree as the following:

```
Inductive tree (V : Type) : Type :=
| E
| T (l : tree V) (v : V) (r : tree V).
Arguments E {V}.
Arguments T {V}.
```

As an example, we can create a simple binary tree with values of type `nat` :

```
Example ex_tree : tree nat :=
  (T (T E 1 E) 2 (T E 1 E)).
(* which represents the tree
          2
        /   \
       1     1
      / \   / \
     E   E E   E                *)
```

(a) (8 points) We want to define some useful operations over binary trees. The map operation on a list applies a function to every element and replaces each element with the result after applying the function. We want to implement `tree_map` as the analagous function over binary trees. For example, the following should be easily provable:

```
Example tree_map_ex1 :
  tree_map (fun v ⇒ v + 1) ex_tree =
    T (T E 2 E) 3 (T E 2 E).
```

Fill in the missing parts of the `tree_map` implementation.

```
Fixpoint tree_map {V W: Type} (f : V → W) (t : tree V) : (tree W) :=
  match t with
  | E ⇒ E
  | T tl v tr ⇒ T (tree_map f tl) (f v) (tree_map f tr)
  end.
```

[Standard Track Only]

(b) (12 points) Now let's define a fold operation over binary trees. Similar to the fold operation for lists, fold for trees should intuitively insert some operation between all elements of a tree. For a list, `fold plus [1;2;3] 0` meant `1 + (2 + (3 + 0))`, but now for a tree we will have that

`tree_fold plus3 (T (T E 1 E) 2 E) 0 = (plus3 (plus3 0 1 0) 2 0) = 3`

where we define `plus3` as a ternary operator:

```
Definition plus3 (x y z : nat) : nat := x + y + z.
```

The implementation of `tree_fold` should make the following example easily provable, where `ex_tree` is the example tree from the previous page.

```
Example tree_fold_ex1 :
  tree_fold plus3 0 ex_tree = 4.
```

Implement the missing parts of the `tree_fold` function.

```
Fixpoint tree_fold {V W : Type} (f : W → V → W → W) (a0 : W) (t : tree V) : W :=
  match t with
  | E ⇒ a0
  | T tl v tr ⇒ f (tree_fold f a0 tl) v (tree_fold f a0 tr)
  end.
```

5. **Defining Inductive Propositions** (16 points total)

This problem asks you to define inductive propositions that work with the type `tree` defined below. (This is the same definition as in problem 5.)

```
Inductive tree (V : Type) : Type :=
| E
| T (l : tree V) (v : V) (r : tree V).
Arguments E {V}.
Arguments T {V}.
```

(a) (4 points) Complete the following definition of an inductive proposition `is_empty` such that `is_empty t` is provable if and only if `t = E`.

```
Inductive is_empty {V : Type} : tree V → Prop :=
| is_empty_E : is_empty E.
```

(b) (12 points) Complete the following definition of an inductive proposition `tree_ex` such that `tree_ex P t` is provable if and only if the tree `t` contains at least one node `(T tl v tr)` such that `P v` holds.

```
Inductive tree_ex {V} (P:V → Prop) : tree V → Prop :=
| te_T : ∀ tl tr v, P v → tree_ex P (T tl v tr)
| te_left : ∀ tl tr v, tree_ex P tl → tree_ex P (T tl v tr)
| te_right : ∀ tl tr v, tree_ex P tr → tree_ex P (T tl v tr)
.
```

6. **Working with Inductive Propositions** (18 points total)

Consider the following inductively defined proposition. Intuitively, `subseq l1 l2` asserts that list `l1` is a *subsequence* of the list `l2`, that is, that all of the elements of the list `l1` appear (not necessarily contiguously) in the same order within `l2`.

```
Inductive subseq {A:Type} : list A → list A → Prop :=
| s_nil : ∀ (ys: list A), subseq [] ys

| s_cons  : ∀ (x:A) (xs:list A) (ys:list A),
    subseq xs ys → subseq (x::xs) (x::ys)

| s_skip : ∀ (xs:list A) (y:A) (ys:list A),
    subseq xs ys → subseq xs (y::ys).
```

For example, we would be able to prove the following:

```
Example example : subseq [1;2] [3;1;4;2].
```

But we would *not* be able to prove this one (because 1 does not follow 2):

```
Example example_fail : subseq [2;1] [3;1;4;2].  (* Not provable! *)
```

(a) (4 points) Which of the following assertions are provable using the definition of `subseq` given above? (Mark all that apply.)

☒ `subseq [] [1]`

☐ `subseq [1] []`

☒ `subseq [1;3] [1;1;3]`

☐ `subseq [1;1;3] [1;3]`

(b) (6 points) In the blanks below, write two distinct *terms*, both of type `subseq [2] [1;2;2]`:

```
Example ans1 : subseq [2] [1;2;2] :=
  s_skip [2] 1 [2;2] (s_cons 2 [] [2] (s_nil [2])).

Example ans2 : subseq [2] [1;2;2] :=
  s_skip [2] 1 [2;2] (s_skip [2] 2 [2] (s_cons 2 [] [] (s_nil []))).
```

(c) (4 points) Consider the following lemma that is provable from the definitions above:

```
Lemma subseq_app_r : ∀ (A:Type) (xs ys1 ys2 : list A),
    subseq xs ys1 →
    subseq xs (ys1 ++ ys2).
```

Mark the checkboxes below to indicate the structure of the proof:

Lemma `subseq_app_r` is most easily proved by induction on:

☐ `xs` ☐ `ys1` ☐ `ys2` ☒ the evidence for `subseq xs ys1`

because the evidence constructed for `subseq xs (ys1 ++ ys2)` will...

- ☐ use the `s_nil` and `s_cons` constructors to mirror the structure of that list.
- ☒ follow exactly the same structure as for `subseq xs ys1`, except that in every step the part of the evidence corresponding to `ys1` has `ys2` appended.
- ☐ follow exactly the same structure as for `subseq xs ys1`, except that in every step the part of the evidence corresponding to `xs` has `ys2` appended.
- ☐ repeatedly use the `s_skip` constructor to skip over the list used for induction and then use the fact that `subseq xs ys1`.

(d) (4 points) Consider the following lemma that is provable from the definitions above:

```
Lemma subseq_app_l : ∀ (A:Type) (xs ys1 ys2 : list A),
    subseq xs ys2 →
    subseq xs (ys1 ++ ys2).
```

Mark the checkboxes below to indicate the structure of the proof:

Lemma `subseq_app_l` is most easily proved by induction on:

☐ `xs` ☒ `ys1` ☐ `ys2` ☐ the evidence for `subseq xs ys2`

because the evidence constructed for `subseq xs (ys1 ++ ys2)` will...

- ☐ use the `s_nil` and `s_cons` constructors to mirror the structure of that list.
- ☐ follow exactly the same structure as for `subseq xs ys2`, except that in every step the part of the evidence corresponding to `ys2` has `ys1` appended to the front.
- ☐ follow exactly the same structure as for `subseq xs ys2`, except that in every step the part of the evidence corresponding to `ys2` has `xs` appended to the front.
- ☒ repeatedly use the `s_skip` constructor to skip over the list used for induction and then use the fact that `subseq xs ys2`.

7. **[Advanced Track Only] Informal Proof** (20 points)

This problem uses the *same* definition of `subseq` as in the previous question. We replicate the definition here for your convenience. Intuitively, `subseq l1 l2` asserts that list `l1` is a *subsequence* of the list `l2`, that is, that all of the elements of the list `l1` appear (not necessarily contiguously) in the same order within `l2`.

```
Inductive subseq {A:Type} : list A → list A → Prop :=
| s_nil : ∀ (ys: list A), subseq [] ys

| s_cons  : ∀ (x:A) (xs:list A) (ys:list A),
    subseq xs ys → subseq (x::xs) (x::ys)

| s_skip : ∀ (xs:list A) (y:A) (ys:list A),
    subseq xs ys → subseq xs (y::ys).
```

Using these definitions, it is possible to prove the following two lemmas:

```
Lemma subseq_app_r : ∀ (A:Type) (xs ys1 ys2 : list A),
    subseq xs ys1 →
    subseq xs (ys1 ++ ys2).

Lemma subseq_app_l : ∀ (A:Type) (xs ys1 ys2 : list A),
    subseq xs ys2 →
    subseq xs (ys1 ++ ys2).
```

On the following page, write a careful *informal* proof of the following fact. The proof uses induction, and we have given you a "skeleton" of the main structure to help you get started. You may use one or both of the lemmas above in your proof. Make sure to state the induction hypothesis explicitly.

```
Lemma subseq_app : ∀ (A:Type) (xs ys ws zs : list A),
    subseq xs ys →
    subseq ws zs →
    subseq (xs ++ ws) (ys ++ zs).
```

Proof

Suppose `subseq xs ys` and `subseq ws zs`. We want to show `subseq (xs ++ ws) (ys ++ zs)`. We proceed by induction on the structure of the evidence for the first hypothesis, and consider the following cases:

- **Case `s_nil`:** The constructor was `s_nil` and we have `xs = []`. Then it suffices to show `subseq ([] ++ ws) (ys ++ zs)`, but that follows by using lemma `subseq_app_l` with the second hypothesis, taking `ys1 = ys`.
- **Case `s_cons`:** The last constructors used was `s_cons` and we have `xs = x::xs'` and `ys = x::ys'` for some `x`,`xs'`, and `ys'`. From the induction hypothesis, we have `subseq (xs'++ ws) (ys'++ zs)`. Observe that `xs ++ ws = (x :: xs') ++ ws = x :: (xs'++ ws)` and similarly for the `ys ++ zs`, so the result follows by applying `s_cons` to the IH.
- **Case `s_skip`:** The last constructors used was `s_cons` and we have `ys = y::ys'` for some `y` and `ys'`. From the induction hypothesis, we have `subseq (xs'++ ws) (ys'++ zs)`. Observe that we have `ys ++ zs = (y :: ys') ++ zs = y :: (ys'++ zs)`, so the result follows by applying `s_skip` to the IH.