CIS 5000: Software Foundations **Midterm I**

October 5, 2023

Solutions

1 (12 points)

Put an X in the *True* or *False* box for each statement, as appropriate.

(a) Any goal state with `True` as one of the assumptions above the line is provable.

☐ True ☒ False

(b) This proposition is provable in Coq with no axioms:

```
forall (P : Prop), ~(P /\ ~P)
```

☒ True ☐ False

(c) This proposition is provable in Coq with no axioms:

```
forall (P : Prop), (P \/ ~P)
```

☐ True ☒ False

(d) Given any goal state where `rewrite H` can be used successfully, `apply H` will also succeed.

☐ True ☒ False

(e) If the current goal goal has the form

```
H: False
--------
False
```

then the `discriminate` tactic will complete the proof of this goal.

☐ True ☒ False

(f) If the current goal state has the form

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

the `injection` followed by `reflexivity` will leave no subgoals.

☐ True ☒ False

(g) Coq's termination checker requires that any `Inductive` type must have at least one non-recursive constructor.

☐ True ☒ False

(h) There are no empty types in Coq. In other words, for any type `A`, there is some Coq expression that has type `A`.

☐ True ☒ False

(i) There is no Coq expression that has type `False` in an empty context.

☒ True ☐ False

(j) In Coq, the propositions `True` and `~False` are logically equivalent—i.e., we can prove `True <-> ~False`.

☒ True ☐ False

(k) For every property of numbers `P : nat -> Prop`, we can construct a boolean function `testP : nat -> bool` such that `testP` reflects `P`.

☐ True ☒ False

(l) The proof object corresponding to an implication `P -> Q` is a function that uses a proof of the proposition `P` to build a proof of the proposition `Q`.

☒ True ☐ False

2 **[Standard Track Only]** (18 points)

What is the type of each of the following Coq expressions? Check one of the listed possibilities. (Check "none of the above" if the expression is typeable but none of the given choices is its type. Check "ill-typed" if the expression does not have a type.)

(a) `1 + 1 = 3`

- ☐ `eq`
- ☐ `False`
- ☐ `false`
- ☒ `Prop`
- ☐ `nat -> nat -> Prop`
- ☐ ill-typed
- ☐ none of the above

(b) `fun (x : nat) => x \/ S x`

- ☐ `Prop`
- ☐ `nat -> Prop`
- ☐ `True`
- ☐ `False`
- ☐ `forall (n : nat), false`
- ☐ `forall (n : nat), False`
- ☒ ill-typed
- ☐ none of the above

(c) `(False, True)`

- ☐ `Prop`
- ☒ `Prop * Prop`
- ☐ `(Prop, Prop)`
- ☐ `False`
- ☐ `false`
- ☐ `X * Y`
- ☐ ill-typed
- ☐ none of the above

(d) `ReflectT (0 < 1)`

- ☐ `Prop`
- ☐ `bool -> Prop`
- ☐ `Prop -> Prop`
- ☐ `reflect (0 < 1)`
- ☐ `reflect (0 < 1) true`
- ☒ `0 < 1 -> reflect (0 < 1) true`
- ☐ ill-typed
- ☐ none of the above

(e) `Union EmptySet`

- ☐ `reg_exp string`
- ☒ `reg_exp string -> reg_exp string`
- ☐ `reg_exp`
- ☐ `reg_exp -> reg_exp`
- ☐ `string`
- ☐ ill-typed
- ☐ none of the above

(The fact that type inference is involved made this question unexpectedly tricky. The were actually three arguably correct answers: b, d, and g. The best answer (in our opinion) was b, because the expression actually does have this type. But g is defensible because what coq will actually print when asked the type of this term is "`reg_exp ?T -> reg_exp ?T`*". And d is defensible because one might imagine that the "string" argument to the* `reg_exp` *type is inferred rather than explicit.)*

(f) `leb`

- ☐ `nat`
- ☐ `nat -> nat`
- ☒ `nat -> nat -> bool`
- ☐ `bool`
- ☐ `nat -> bool`
- ☐ `nat -> nat -> Prop`
- ☐ ill-typed

☐ none of the above

(g) `forall (x : nat), leb x x`

☐ `nat`

☐ `nat -> nat`

☐ `nat -> nat -> bool`

☐ `bool`

☐ `nat -> bool`

☐ `nat -> nat -> Prop`

☒ ill-typed

☐ none of the above

(h) `fun n => forall m, eqb m n = true`

☐ `nat`

☐ `nat -> bool`

☐ `nat -> nat -> bool`

☐ `Prop`

☒ `nat -> Prop`

☐ `nat -> nat -> Prop`

☐ ill-typed

☐ none of the above

3 (18 points) For each of the types below, write a Coq expression that has that type, or else write "uninhabited" if there are no such expressions.

(a) `[1;2] =~ App (Char 1) (Union (Char 2) (Char 3))`

Answer: Example: `MApp [1] (Char 1) [2] (Union (Char 2) (Char 3)) (MChar 1) (MUnionL [2] (Char 2) (Char 3) (MChar 2))`

This one was unintentionally difficult: much partial credit was given.

(b) `[1;2] =~ Star (Char 1)`

Answer: uninhabited

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

Answer: uninhabited

(d) `forall X Y, X -> (X -> Y) -> Y`

Answer: fun X Y (x:X) (f:X->Y) => f x

(e) `4 <= 3`

Answer: uninhabited

(f) `(4 <= 3) -> (4 <= 4)`

Answer: le_S 4 3

4 In this problem, we will work with two different implementations of the `find` function, which has this type:

```
list nat -> (nat -> bool) -> option nat
```

Invoking `find l f` should return the first element in the list `l` that satisfies the predicate `f` (wrapped in `Some`), or else `None` if there aren't any elements that satisfy `f`. For example:

```
find [1; 4; 5] even = Some 4.

find [1; 2; 4] (fun _ => true) = Some 1.

find [1; 3] even = None.

find [] (fun _ => true) = None.
```

(a) (5 points) First, let's implement `find` as a simple recursive function (without calling any other functions in its body). Fill in the skeleton below.

```
Fixpoint find (l : list nat) (f : nat -> bool) : option nat :=
```

Answer:

```
match l with
| [ ] => None
| (h :: t) => if f h then Some h else find t f
end.
```

(b) (7 points)

Recall the definition of `fold`:

```
Fixpoint fold {X Y: Type} (f : X->Y->Y) (l : list X) (b : Y) : Y :=
  match l with
  | nil => b
  | h :: t => f h (fold f t b)
  end.
```

We can implement `find` using `fold` as something like this:

```
Definition find (l : list nat) (f : nat -> bool) : option nat :=
  fold myfunction l None.
```

Mark each of the following potential replacements for `myfunction` above correct or incorrect. If incorrect, provide a list `l` such that `find l even` would return the wrong answer if this replacement were used.

`myfunction =`

```
(fun h acc => Some h)
```

☐ Correct.

☒ Incorrect. Counterexample: `l = [1]`

`myfunction =`

```
(fun h acc => if f h then Some h else None)
```

☐ Correct.

☒ Incorrect. Counterexample: `l = [1; 2]`

`myfunction =`

```
(fun h acc => if f h then Some h else acc)
```

☒ Correct.

☐ Incorrect. Counterexample: `l =`

5 This problem asks you to translate informal mathematical ideas expressed in English into formal ones in Coq.

(a) (4 points) A *composite* number is one that can be formed by multiplying two numbers that are each strictly greater than one.

For example, 4 is composite ($4 = 2 \times 2$), but 3 is not.

```
Definition composite (n : nat) : Prop :=
```

Answer:

```
exists p q,
  p > 1 /\ q > 1 /\ p * q = n.
```

(b) (4 points) A *prefix* of a list is a sub-list that occurs at the beginning of a larger list. For example, these are all the prefixes of `[1;2;3]` (i.e., all the lists `s` such that `prefix s [1;2;3]`):

```
[]
[1]
[1;2]
[1;2;3]
```

Complete the following inductive definition so that `prefix s l` is provable exactly when `s` is a prefix of `l`.

```
Inductive prefix {X : Type} : list X -> list X -> Prop :=
```

Answer:

```
| prefix_nil  : forall l,      prefix [] l
| prefix_cons : forall t l h, prefix t l -> prefix (h::t) (h::l).
```

(c) (5 points) Given lists `l1` and `l2` and item `x`, we say that `inserted x l1 l2` when `l2` is just the list `l1` with one occurrence of the element `x` inserted somewhere inside it.

For example:

```
inserted 42 [1;2;3] [42;1;2;3]
inserted 42 [1;2;3] [1;2;42;3]
inserted 1 [1;2;3] [1;2;3;1]
inserted 1 [1;2;3] [1;1;2;3]
```

Complete the following inductive definition. (It should have two cases: one for when `x` appears at the front of `l2` and one for when `x` is inserted somewhere in the tail.) *Later:*

```
Inductive inserted {X : Type} : X -> list X -> list X -> Prop :=
| here : forall (x : X) (l1 l2 : list X),
    inserted x l1 (x :: l2)
| later : forall (x : X) (t1 t2 : list X) (h : X),
    inserted x t1 t2 ->
    inserted x (h::t1) (h::t2).
```

(d) (5 points) A list `l1` is a *permutation* of another list `l2` if `l1` and `l2` have the same elements (with each element occurring the same number of times), possibly in different orders.

For example, the following lists (among others) are permutations of `[1;1;2;3]`:

```
[1;1;2;3]
[2;1;3;1]
[3;2;1;1]
[1;3;2;1]
```

On the other hand, `[1;2;3]` is *not* a permutation of `[1;1;2;3]`.

Here is one way to define the concept of permutation precisely:

- The empty list is a permutation of itself.
- If two lists are permutations of each other, then inserting the same element at an arbitrary position in each list yields longer lists that are again permutations.

Use the `inserted` relation from part (c) to formalize this definition as an inductive relation.

```
Inductive perm {X : Type} : list X -> list X -> Prop :=

| empty : perm [] []
| more : forall x l1 l2 l1' l2',
    perm l1 l2 ->
    inserted x l1 l1' ->
    inserted x l2 l2' ->
    perm l1' l2'.
```

6 (12 points) For each of the following propositions, check "**not provable**" if it is not provable (without additional axioms), "**needs induction**" if it is provable only using induction, or "**easy**" if it is provable without using induction and without additional lemmas.

(a) `In 3 [1;2;3]`

☐ not provable ☐ needs induction ☒ easy

(b) `forall x, In x [1;2;3]`

☒ not provable ☐ needs induction ☐ easy

(c) `forall s, In 3 ([1;2;3] ++ s)`

☐ not provable ☐ needs induction ☒ easy

(d) `forall s, In 3 (s ++ [1;2;3])`

☐ not provable ☒ needs induction ☐ easy

(e) `exists s, In 3 (s ++ [1;2;3])`

☐ not provable ☐ needs induction ☒ easy

(f) `forall x, In x [1;2;3] -> In x [3;2;1]`

☐ not provable ☐ needs induction ☒ easy

(g) `forall x s, In x s -> In x ([1;2;3] ++ s)`

☐ not provable ☐ needs induction ☒ easy

(h) `exists (x y : list nat), x ++ y = y ++ x`

☐ not provable ☐ needs induction ☒ easy

(i) `forall n, pred n <= n`

☐ not provable ☐ needs induction ☒ easy

(j) `forall x y z, x + (y + z) = (x + y) + z`

☐ not provable ☒ needs induction ☐ easy

(k) `forall P : Prop, (P /\ ~P) -> True`

☐ not provable ☐ needs induction ☒ easy

(l) `forall P : Prop, P \/ P`

☒ not provable ☐ needs induction ☐ easy

7 **[Advanced Track Only]** (18 points)

Suppose we define a *lexicographic ordering* relation on lists of numbers as follows:

```
Inductive listlt : list nat -> list nat -> Prop :=
| listlt_empty : forall x2,
    listlt [] x2
| listlt_head : forall h1 h2 t1 t2,
    h1 < h2 -> listlt (h1::t1) (h2::t2)
| listlt_tail : forall h t1 t2,
    listlt t1 t2 ->
    listlt (h::t1) (h::t2).
```

For example:

```
listlt [1] [2;3]
listlt [1] [1;2;3]
listlt [1;1] [1;2;3]
```

Fill in the missing parts in the following informal proof that this ordering is transitive.

```
Theorem listlt_trans : forall x y z,
    listlt x y -> listlt y z -> listlt x z.
```

Proof: By induction on the first of the two given derivations. Specifically, we prove, by induction on a derivation of `listlt x y` that, for all `z` and for any derivation of `listlt y z`, we have `listlt x z`.

Case 1: Suppose the first derivation ends with `listlt_empty`.

(Complete this case of the proof)

In this case, the list `x` is empty. Then `listlt x z` follows immediately by `listlt_empty`.

Case 2: Suppose the first derivation ends with `listlt_head`, with

```
x = h1 :: t2
y = h2 :: y2
h1 < h2.
```

(The rest of this case is omitted; leave it blank.)

Case 3: Suppose the first derivation ends with `listlt_tail`, with

```
x = h :: t1
y = h :: t2
listlt t1 t2,
```

and suppose we are given the following induction hypothesis:

(Fill in the IH...) for any `z` and any derivation of `listlt t2 z`, we have `listlt t1 z`.

(Complete this case of the proof...) Again, consider the possible forms of the second derivation.

Subcase 3a: Suppose the first derivation ends with `listlt_empty`. Again, this cannot happen (`y` cannot be both `[]` and `h2::t2`).

Subcase 3b: Suppose the first derivation ends with `listlt_head`, with

```
z = h :: t3
h < h3.
```

Then `listlt x z` is immediate by `listlt_tail`.

Subcase 3c: Suppose the first derivation ends with `listlt_tail`, with

```
z = h :: t3.
```

By the IH, we have `listlt t1 t3`, from which it follows that `listlt x z`, by `listlt_tail`.

For Reference

```
Definition pred (n : nat) : nat :=
  match n with
  | O => O
  | S n' => n'
  end.

Fixpoint leb (n m : nat) : bool :=
  match n with
  | O => true
  | S n' =>
      match m with
      | O => false
      | S m' => leb n' m'
      end
  end.

Inductive le : nat -> nat -> Prop :=
  | le_n (n : nat)                : le n n
  | le_S (n m : nat) (H : le n m) : le n (S m).
Notation "n <= m" := (le n m).

Definition lt (n m : nat) := le (S n) m.
Notation "n < m" := (lt n m).

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

Inductive list (X:Type) : Type :=
  | nil
  | cons (x : X) (l : list X).

Fixpoint In {A : Type} (x : A) (l : list A) : Prop :=
  match l with
  | [] => False
  | x' :: l' => x' = x \/ In x l'
  end.

Inductive reflect (P : Prop) : bool -> Prop :=
  | ReflectT (H :   P) : reflect P true
  | ReflectF (H : ~ P) : reflect P false.
```

```
Inductive True : Prop :=
  | I : True.

Inductive False : Prop := .
```

```
Inductive reg_exp (T : Type) : Type :=
  | EmptySet
  | EmptyStr
  | Char (t : T)
  | App (r1 r2 : reg_exp T)
  | Union (r1 r2 : reg_exp T)
  | Star (r : reg_exp T).

Arguments EmptySet {T}.
Arguments EmptyStr {T}.
Arguments Char {T} _.
Arguments App {T} _ _.
Arguments Union {T} _ _.
Arguments Star {T} _.

Inductive exp_match {T} : list T -> reg_exp T -> Prop :=
  | MEmpty : [] =~ EmptyStr
  | MChar x : [x] =~ (Char x)
  | MApp s1 re1 s2 re2
             (H1 : s1 =~ re1)
             (H2 : s2 =~ re2)
           : (s1 ++ s2) =~ (App re1 re2)
  | MUnionL s1 re1 re2
                (H1 : s1 =~ re1)
              : s1 =~ (Union re1 re2)
  | MUnionR re1 s2 re2
                (H2 : s2 =~ re2)
              : s2 =~ (Union re1 re2)
  | MStar0 re : [] =~ (Star re)
  | MStarApp s1 s2 re
                 (H1 : s1 =~ re)
                 (H2 : s2 =~ (Star re))
               : (s1 ++ s2) =~ (Star re)

  where "s =~ re" := (exp_match s re).
```