CIS 500: Software Foundations **Midterm I**

October 12, 2021

Solutions

1 (8 points) Put an X in the *True* or *False* box for each statement, as appropriate.

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

```
forall (f: A -> A) (x y: A), x = y -> f x = f y.
```

☒ True ☐ False

(b) `[] =~ Star re` is provable for every `re`. (The definition of `=~` can be found in the "For Reference" section at the end.)

☒ True ☐ False

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

```
forall A (f: A -> A) (x y: nat), f x = f y -> x = y.
```

☐ True ☒ False

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

```
False -> False.
```

☒ True ☐ False

(e) The result of `Compute (In 42 [1;2])` is `False`.

☐ True ☒ False

(f) Functions defined in Coq via `Fixpoint` must terminate on all inputs, but functions defined with `Definition` need not always terminate.

☐ True ☒ False

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

☐ True ☒ False

(h) There exists a proposition `P` such that the proposition `~~P <-> P` is provable (with no additional axioms).

☒ True ☐ False

2 **[Standard Track Only]** (10 points)

(a) How many subgoals will we have after running the tactic `inversion H`?

```
H: [a; b] = []
--------------------
2 = 2
```

- ☐ Tactic fails.
- ☒ 0 (solves the goal)
- ☐ 1
- ☐ 2
- ☐ 3

(b) How many subgoals will we have after running the tactic `apply H`?

```
P, Q, R: Prop
H: P -> Q -> R
--------------------
R
```

- ☐ Tactic fails.
- ☐ 0 (solves the goal)
- ☐ 1
- ☒ 2
- ☐ 3

(c) How many subgoals will we have after running the tactic `apply H in H1`?

```
P, Q, R: Prop
H: P -> Q -> R
H1: P
--------------------
R
```

- ☐ Tactic fails
- ☐ 0 (solves the goal)
- ☐ 1
- ☒ 2
- ☐ 3

(d) How many subgoals will we have after running the tactic `induction H`? (The definition of `le` can be found in the "For Reference" section at the end.)

```
n, m: nat
H: lt n m
--------------------
le n m
```

☐ Tactic fails

☐ 0 (solves the goal)

☐ 1

☒ 2

☐ 3

(e) How many subgoals will we have after running the tactic `apply (le_S n n (le_n n))`?

```
n: nat
--------------------
le n (S n)
```

☐ Tactic fails

☒ 0 (solves the goal)

☐ 1

☐ 2

☐ 3

$\boxed{3}$ **[Standard Track Only]** (15 points) What is the type of each of the following Coq expressions? (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) `4 <= 3`

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

(b) `forall (A : Type) (m n : A), m = n \/ m <> n`

- ☐ `forall (A : Type) (m n : A), Prop`
- ☐ `forall (A : Type) A -> A -> Prop`
- ☐ `fun (A : Type) => fun (m n : A) => m =? n`
- ☒ `Prop`
- ☐ `True`
- ☐ `False`
- ☐ ill-typed
- ☐ none of the above

(c) `fun (x : nat) => False`

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

(d) `forall (m : nat), m * m`

- ☐ `Prop`
- ☐ `Prop * Prop`
- ☐ `(nat,nat)`
- ☐ `False`
- ☐ `false`
- ☐ `nat -> nat`
- ☐ `fun (m : nat) => nat`
- ☒ ill-typed
- ☐ none of the above

(e) `beq_nat 3`

- ☐ `(nat,nat)`
- ☐ `bool`
- ☐ `Prop`
- ☒ `nat -> bool`
- ☐ `nat -> Prop`
- ☐ ill-typed
- ☐ none of the above

(f) `fun (P Q : Prop) => P -> Q`

- ☐ `(nat,nat)`
- ☐ `bool`
- ☐ `Prop`
- ☐ `Prop -> Prop`
- ☒ `Prop -> Prop -> Prop`
- ☐ `forall (P Q : Prop), Prop`
- ☐ ill-typed
- ☐ none of the above

(g) `fun (m : nat) (E : 0 <= m) => le_S 0 m E`

- ☐ `Prop`
- ☐ `nat -> Prop`
- ☐ `Prop -> Prop`
- ☐ `forall (m:nat), Prop -> Prop`
- ☒ `forall (m:nat), 0 <= m -> 0 <= S m`
- ☐ `forall (m:nat), 0 <= m -> Prop`
- ☐ ill-typed
- ☐ none of the above

$\boxed{4}$ (15 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) `nat -> (nat -> bool)`

Answer: Example: `leb`

(b) `forall (X Y : Type), list X -> list Y`

Answer: Example: `fun X Y (l : list X) => []`

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

Answer: Example: `fun X Y (a : X) (f : X->X->Y) => f a a`

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

Answer: Uninhabited

(e) `Prop -> bool`

Answer: Example: `fun (P : Prop) => true`

(f) `In 2 [1;1;1]`

Answer: Uninhabited

(g) `ev 1`

Answer: Uninhabited

(h) `forall n : nat, ev n -> ev (S (S n))`

Answer: Example: `ev_SS`

(i) `(nat -> nat) -> nat`

Answer: Example: fun f => (f 0)

$\boxed{5}$ (12 points) The higher-order function `fold_left`...

```
Fixpoint fold_left {A B} (f: B -> A -> B) (a: list A) (b: B) : B :=
  match a with
  | [] => b
  | h :: ts => fold_left f ts (f b h)
  end.
```

... is quite versatile — in fact we can easily define many commonly used functions non-recursively, just by applying `fold_left` to appropriate arguments. For example this is how we can define `map` using `fold_left`:

```
Definition map {A B} (f: A -> B) (a: list A) : list B :=
  fold_left (fun acc e => acc ++ [f e]) a [].
```

Define the following functions using `fold_left`.

(a) Keep the elements of the input list for which the predicate `f` yields `true`.

Example: `filter evenb [1;2;3;4] = [2;4]`

```
Definition filter {A} (f: A -> bool) (a: list A) :=
    fold_left

        (fun acc e => if f e then acc ++ [e] else acc) a [].
```

(b) From a list of pairs, return a pair of lists.

Example: `unzip [(1, true); (2, false); (3, true)] = [1;2;3] [true; false; true]`

```
Definition unzip {X Y} (l: list (X*Y)) : (list X * list Y) :=
    fold_left

            (fun acc e =>
               match acc with
               | (a, b) => (a ++ [fst e], b ++ [snd e])
               end)
            l ([],[]).
```

(c) Apply a predicate `f` on each element of a list and return a pair of lists; if `f` is true for a given element, put it on the left list, otherwise put it on the right list.

Example: `split evenb [1;2;3;4] = ([2;4], [1;3])`

```
Definition split {X} (l: list X) (f: X -> bool) : (list X * list X) :=
  fold_left
          (fun acc e =>
             match acc with
             | (a, b) => if f e then (a ++ [e], b) else (a, b ++ [e])
             end)
          l ([], []).
```

6 (12 points) An expression in Gallina is said to be *canonical* if it cannot be simplified. For example, these expressions are canonical

```
O
S O
S (S O)
true
[true]
```

while these are not:

```
O + 1
negb true
[true] ++ []
(fun (x:nat) => true) 3
```

Note that the type `bool` has two canonical members, while `nat` has infinitely many.

The same notion of "canonical member" also works for expressions whose types involve `Prop`. For example, given the definition of the binary `<=` relation from the IndProp chapter

```
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).
```

the proposition `1<=2` has one canonical member, namely

```
le_S 1 1 (le_n 1)
```

while the proposition `1<=0` is empty.

Each sub-question on the next page presents an inductively defined property `P` of natural numbers and asks you to list the canonical members of `P n` for some `n`. If `P n` has infinitely many canonical members, write "infinite." If it has no members, write "empty."

6.1 Define `P` as follows:

```
Inductive P : nat -> Prop :=
| A : P 0
| B : P 1
| C : P 0.
```

What are the canonical members of `P 0`? (List all of them in the space below.)

```
Check A : P 0.
Check C : P 0.
```

$\boxed{6.2}$ Define `P` as follows:

```
Inductive P : nat -> Prop :=
| A : P 0
| B (n : nat) : P n.
```

What are the canonical members of `P 0`?

```
Check A : P 0.
Check B 0 : P 0.
```

$\boxed{6.3}$ Define `P` as follows:

```
Inductive P: nat -> Prop :=
| B (n:nat) (H: P n) : P (S n).
```

What are the canonical members of `P 1`?

```
(* Empty! *)
```

$\boxed{6.4}$ Define `P` as follows:

```
Inductive P: nat -> Prop :=
| A : P 1
| B  (n:nat) (H: P (S n)) : P (S n).
```

What are the canonical members of `P 1`?

```
(* Infinite! *)
Check A : P 1.
Check B 0 A : P 1.
Check B 0 (B 0 A) : P 1.
Check B 0 (B 0 (B 0 A)) : P 1.
Check B 0 (B 0 (B 0 (B 0 A))) : P 1.
```

$\boxed{6.5}$ Define `P` as follows:

```
Inductive P : nat -> Prop :=
| A : P 1
| B (n : nat)  (H : n <> n) : P n.
```

What are the canonical members of `P 1`?

```
Check A : P 1.
```

$\boxed{6.6}$ Define `P` as follows:

```
Inductive P : nat -> Prop :=
| A (n : nat) (H0 : n <= 1) : P n.
```

What are the canonical members of `P 1`?

```
Check A 1 (le_n 1) : P 1.
```

$\boxed{7}$ (12 points) In this problem we will be working with the following definition of single-variable polynomials over the natural numbers.

```
Inductive Poly :=
| Var
| Const (a: nat)
| Sum (a b: Poly)
| Prod (a b: Poly).
```

The *associative law* for addition says that changing a subexpression of the form $x + (y + z)$ to $(x + y) + z$ or vice versa yields an equivalent polymomial.

Your job is to complete the definition of the inductive relation `reassoc`, where `reassoc p1 p2` means that `p1` and `p2` are "equivalent modulo associativity of plus." For example,

```
reassoc (Prod (Sum (Const 0)
                   (Sum (Const 1) (Const 2)))
              (Const 3))
        (Prod (Sum (Sum (Const 0) (Const 1))
                        (Const 2))
              (Const 3)).
(* i.e.,            (0 + (1 + 2)) * 3
   is equivalent to ((0 + 1) + 2) * 3    *)
```

We've given you a few of the constructors; you supply the rest.

```
Inductive reassoc : Poly -> Poly -> Prop :=
| refl : forall p,
    reassoc p p
| trans : forall p1 p2 p3,
    reassoc p1 p2 ->
    reassoc p2 p3 ->
    reassoc p1 p3
| sum : forall p1 p1' p2 p2',
    reassoc p1 p1' ->
    reassoc p2 p2' ->
    reassoc (Sum p1 p2) (Sum p1' p2')

| prod : forall p1 p1' p2 p2',
    reassoc p1 p1' ->
    reassoc p2 p2' ->
    reassoc (Prod p1 p2) (Prod p1' p2')
| assoc : forall p1 p2 p3,
    reassoc (Sum p1 (Sum p2 p3)) (Sum (Sum p1 p2) p3)
| symm : forall p1 p2,
    reassoc p1 p2 ->
    reassoc p2 p1.
```

$\boxed{8}$ **[Standard Track Only]** (6 points)

Let's translate some English statements about polynomials into Coq theorems. First, some definitions...

An *evaluation* function for polynomials can be written as follows:

```
Fixpoint eval(p: Poly)(x: nat): nat :=
  match p with
  | Var => x
  | Const n => n
  | Sum a b => eval a x + eval b x
  | Prod a b => eval a x * eval b x
  end.
```

A polynomial is *constant* if it always yields the same result, no matter the value of the variable:

```
Definition constant (p : Poly) : Prop :=
  exists r, forall n, eval p n = r.
```

Two polynomials are *equivalent* if they yield the same result for every value of the variable:

```
Definition equiv (p1 p2 : Poly) : Prop :=
  forall n, eval p1 n = eval p2 n.
```

The *degree* of a polynomial is the highest power of the variable that appears in its "fully multiplied out" form. For example $x * x + x + 2 + x * x * 3$ and $(x + 1) * (x + 2)$ both have degree 2. Here is a definition of the `degree` function.

```
Fixpoint degree(p: Poly): nat :=
  match p with
  | Var => 1
  | Const a => 0
  | Sum a b => max (degree a) (degree b)
  | Prod a b => degree a + degree b
  end.
```

(a) Write a theorem stating that "degree-zero polynomials are constant and vice versa." (No need to prove it—just state the theorem.)

```
Theorem deg0_constant : forall (p : Poly),

    degree p = 0 <-> constant p.
```

(b) Write a theorem stating that "Every polynomial of degree at most 1 is equivalent to one of the form $ax + b$."

```
Theorem nf : forall (p : Poly),

        degree p <= 1
    <-> exists a b, equiv p (Sum (Prod (Const a) Var) (Const b)).
```

9 **[Advanced Track Only]** (14 points)

Recall the definition of `In`

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

and the following lemma from `Logic.v`:

```
Lemma In_app_iff : forall A l l' (a:A),
  In a (l++l') <-> In a l \/ In a l'.
```

Give a careful informal proof of the *left-to-right* direction of this theorem. If your proof goes by induction, make sure to state any induction hypotheses *explicitly*.

```
Lemma In_app_iff : forall A l l' (a:A),
  In a (l++l') -> In a l \/ In a l'.

Proof: By induction on the list l.

Base case: l = [].  In this case, l++l' = l', so

     In a (l++l') = In a l',

   and the result is immediate.

Induction case: l = h::t, with induction hypothesis

  IH = In a (t++l') -> In a l \/ In a l'.

  By the definition of In, we know

          In a (l++l')
    i.e., In a ((h::t)++l'))
    i.e., In a (h::(t++l'))
    i.e., a=h \/ In a (t++l')

    Suppose a=h.  Then

            a=h \/ In a t
      i.e., In a (h::t)
      i.e., In a l

    and the result is immediate.

    The other possibility is In a (t++l'), and the result is
    again immediate.
```

10 **[Advanced Track Only]** (17 points)

Recall the `Fixpoint` definition of list membership from the `Logic` chapter:

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

If we define a simple datatype of binary trees...

```
Inductive tree (A : Type) : Type :=
| leaf
| node (label : A) (ll rr : tree A).
```

... we can give a similar definition of "tree membership" like this:

```
Fixpoint TIn {A : Type} (x : A) (t : tree A) : Prop :=
  match t with
  | leaf _ => False
  | node _ a ll rr => a = x \/ TIn x ll \/ TIn x rr
  end.
```

Next, let's define a function `squish` that flattens a tree into the list of its labels:

```
Fixpoint squish {A : Type} (t : tree A) : list A :=
  match t with
  | leaf _ => []
  | node _ a ll rr => [a] ++ (squish ll ++ squish rr)
  end.
```

Now we can state a theorem saying, informally, that "squishing commutes with membership"—i.e., that a given element `x` is a member of a tree `t` iff `x` is a member of `squish t`.

```
Theorem TIn_squish : forall A (x : A) (t : tree A),
    In x (squish t) -> TIn x t.
```

On the next page, give a careful informal proof of this theorem. If your proof goes by induction, make sure to state any induction hypotheses *explicitly*.

```
Theorem TIn_squish : forall A (x : A) (t : tree A),
    In x (squish t) -> TIn x t.
```

Proof:

```
By induction on t.

  - Base case: t = leaf.  Then squish t is [] by definition, and
      TIn A x t = In A x (squish t) = False.
    The result is immediate.

  - Induction case: We are given
        t = node a ll rr
        IH1: In A x (squish ll) -> TIn A x ll
        IH2: In A x (squish rr) -> TIn A x rr

    By the definition of squish,

        squish t = [a] ++ (squish ll ++ squish rr).

    Reason as follows:

          In A x (squish t)
      <-> In A x [a] \/ In A x ll \/ In A x rr.
            (by In_app_iff, twice)
      ->  In A x [a] \/ TIn A x (squish ll) \/ TIn A x (squish rr).
            (by IH1 and IH2)
      ->  x = a \/ False \/ TIn A x (squish ll) \/ TIn A x (squish rr).
            (by the definition of In)
      <-> x = a \/ TIn A x (squish ll) \/ TIn A x (squish rr).

    By the definition of TIn A x (node a ll rr),

      In A x (squish t) -> x = TIn A x t

    as required.
```

For Reference

```
Fixpoint beq_nat(a b: nat): bool :=
  match a, b with
  | S a', S b' => beq_nat a' b'
  | 0, 0 => true
  | _, _ => false
  end.

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

Fixpoint fold_left {A B} (f: B -> A -> B) (a: list A) (b: B) : B :=
  match a with
  | [] => b
  | h :: ts => fold_left f ts (f b h)
end.


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

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

```
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).

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).
```