

CIS 500 — Software Foundations

Midterm I

(Standard version)

September 30, 2014

Name: \_\_\_\_\_

Pennkey: \_\_\_\_\_

Scores:

1	
2	
3	
4	
5	
6	
7	
8	
Total (90 max)	

1. (10 points) Circle True or False for each statement.

(a) All functions defined in Coq via `Fixpoint` must terminate on all inputs.

True      False

(b) The proof of an implication  $P \rightarrow Q$  is a function that uses a proof of the proposition  $P$  to produce a proof of the proposition  $Q$ .

True      False

(c) The proposition `true = false` is provable in Coq.

True      False

(d) Given a function `f` of type `nat -> bool`, it is possible to define a proposition that holds when `f` returns `true` for all natural numbers.

True      False

(e) 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

(f) If  $H : \text{true} = \text{false}$  is a current assumption, then the tactic `inversion H` will solve any goal.

True      False

(g) If  $H : S\ x = S\ (S\ y)$  is a current assumption, then the tactic `inversion H` will solve any goal.

True      False

(h) If  $H : x <> y$  is a current assumption, then the tactic `inversion H` will solve any goal.

True      False

(i) If the goal is  $A \wedge B$ , then the tactic `split` will produce two subgoals, one for  $A$  and one for  $B$ .

True      False

(j) If  $H : x1 :: y1 = x2 :: y2$  is a current assumption, then we know that  $x1$  is equal to  $x2$ .

True      False

2. (10 points) Write the type of each of the following Coq expressions, or write “ill-typed” if it does not have one. (The references section contains the definitions of some of the mentioned functions and propositions.)

(a) `beq_nat 3 4`

(b) `3=4`

(c) `forall (X:Type), forall (x:X), x = x`

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

(e) `fun (x:nat) => x :: x`

3. (8 points) For each of the types below, write a Coq expression that has that type or write “Empty” if there are no such expressions. (The references section contains the definitions of some of the mentioned functions and propositions.)

(a) `forall (X:Type), list X -> nat`

(b) `Prop`

(c) `beautiful 8`

(d) `forall (X:Prop), X -> ~~X`

4. (12 points) For each of the given theorems, which set of tactics is needed to prove it besides `intros` and `reflexivity`? If more than one of the sets of tactics will work, choose the smallest set. Note that each proof should be completed directly, without the help of any lemmas.

(a) Theorem `mult_0_1` : forall n:nat, 0 \* n = 0.

- i. `induction` and `rewrite`
- ii. `rewrite` and `simpl`
- iii. `inversion`
- iv. no additional tactics are necessary

(b) Lemma `plus_assoc` : forall n m p : nat, n + (m + p) = (n + m) + p.

- i. `simpl` and `rewrite`
- ii. `simpl`, `rewrite` and `induction`
- iii. `rewrite`
- iv. no additional tactics are necessary

(c) Lemma `and_assoc` : forall P Q R : Prop, P /\ (Q /\ R) -> (P /\ Q) /\ R.

- i. `inversion`
- ii. `rewrite`, `induction`, and `inversion`
- iii. `inversion`, `split`, and `apply`
- iv. no additional tactics are necessary

(d) Theorem `ble_plus` : forall n m p : nat, ble\_nat n m = true ->  
ble\_nat (p + n) (p + m) = true.

- i. `simpl`, `apply`, and `destruct n`
- ii. `simpl`, `apply`, and `induction p`
- iii. `simpl`, `apply`, and `induction n`
- iv. no additional tactics are necessary

5. (17 points) An alternate way to encode lists in Coq is the `jlist` type, shown below.

```
Inductive jlist (X:Type) : Type :=
  | j_nil : jlist X
  | j_one : X -> jlist X
  | j_app : jlist X -> jlist X -> jlist X.
```

(\* Make the type parameter implicit \*)

Arguments j\_nil {X}.

Arguments j\_one {X} \_.

Arguments j\_app {X} - \_.

We can convert a `jlist` to a regular `list` with the following function:

```
Fixpoint to_list {X : Type} (jl : jlist X) : list X :=
  match jl with
  | j_nil => []
  | j_one x => [x]
  | j_app j1 j2 => to_list j1 ++ to_list j2
  end.
```

- (a) Note that there may be multiple `jlists` that represent the same `list`. Demonstrate this fact by giving definitions of `example1` and `example2` such that the Lemma below (`distinct_jlists_to_same_list`) is provable (there is no need to prove it).

```
Definition example1 : jlist nat :=
```

```
Definition example2 : jlist nat :=
```

```
Lemma distinct_jlists_to_same_list :
  example1 <> example2 /\ (to_list example1) = (to_list example2).
```

- (b) It is also possible to define most list operations directly on the `jlist` representation. Complete the following function for mapping over a `jlist`:

```
Fixpoint j_map {X Y :Type} (f : X -> Y) (x : jlist X) : jlist Y :=
```

- (c) What is the type of the expression `j_one` ?

- (d) What is the type of the expression `j_map (fun (x:nat) => beq_nat x 0)` ?

- (e) Your `j_map` function from part (b) should satisfy the following correctness lemma that states that it agrees with the `list` map operation. (The `list` map function is shown in the references.) The proof of this lemma for our definition of `j_map` is shown below. This proof uses an auxiliary lemma (`map_app`), not shown.

```
Lemma j_map_correct : forall (X:Type) (Y:Type) (f : X -> Y) (l:jlist X),
  to_list (j_map f l) = map f (to_list l).
```

Proof.

```
intros X Y f l. induction l as [|x|l1 IH1 l2 IH2].
```

Case "j\_nil".

```
  simpl. reflexivity.
```

Case "j\_one".

```
  simpl. reflexivity.
```

Case "j\_app".

```
  simpl. rewrite IH1. rewrite IH2. apply map_app.
```

Qed.

The `j_app` case of the `j_map` correctness proof makes use of two different induction hypotheses, called `IH1` and `IH2`. Circle the correct statement of `IH1` used in this case of the proof.

- i. `IH1: to_list (j_app l1 l2) = map f (to_list (j_app l1 l2))`
- ii. `IH1: to_list (j_map f l1) = map f (to_list l1)`
- iii. `IH1: forall l1:jlist X. to_list (j_map f l1) = map f (to_list l1)`
- iv. `IH1: forall l2:jlist X. to_list (j_app l1 l2) = map f (to_list (j_app l1 l2))`

Circle the statement of the lemma `map_app`, necessary to complete the `j_app` case of the `j_map` correctness proof.

- i. `Lemma map_app : forall X Y (f:X -> Y) l1 l2,`  
 `j_map f (j_app l1 l2) = j_app (j_map f l1) (j_map f l2)`
- ii. `Lemma map_app : forall X Y (f:X -> Y) l1 l2,`  
 `map f (l1 ++ l2) = j_map f (j_app l1 l2)`
- iii. `Lemma map_app : forall X Y (f:X -> Y) l1 l2,`  
 `map f l1 ++ map f l2 = j_app (j_map f l1) (j_map f l2)`
- iv. `Lemma map_app : forall X Y (f:X -> Y) l1 l2,`  
 `map f l1 ++ map f l2 = map f (l1 ++ l2).`

6. (13 points) In this question, we'll consider two different implementations of the same list function—one as an inductively defined relation and one as a `Fixpoint`.

- (a) The function `f_repeat` takes an element `x` and a number `n` and returns a list containing `n` copies of the element. For example:

```
f_repeat true 3 = [true; true; true]
f_repeat 4 0 = []
```

Complete the `Fixpoint` definition of `f_repeat`.

```
Fixpoint f_repeat {X : Type} (x : X) (n : nat) : list X :=
```

- (b) Similarly, the relation `r_repeat` is a three place relation that holds between an element `x`, a number `n`, and a list `xs` if and only if `xs` is the list obtained by repeating the element `n` times. For example, the following are provable instances of `r_repeat`.

```
r_repeat true 3 [true; true; true]
r_repeat 4 0 []
```

Complete an `Inductive` definition of `r_repeat`. Note, your answer must not use `f_repeat`.

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

- (c) Suppose we want to show the equivalence between the functional definition of repetition and the relational specification. As part of that, we should prove the following lemma:

```
Lemma repeat_f_to_r : forall X x n (l : list X),
  f_repeat x n = l -> r_repeat x n l.
```

An ill-advised proof of this lemma *might* start as follows:

Proof.

```
intros X x n l H. induction n as [|n'].
Case "0".
  admit. (* skipping base case for now. *)
Case "n = S n'".
  destruct l as [|x0 l0].
  SCase "l=[]". simpl in H. inversion H.
  SCase "l=x0 :: l0".
```

At this point, the proof state looks like the following:

```
SCase := "l=x0 :: l0" : String.string
Case := "S n'" : String.string
X : Type
x : X
n' : nat
x0 : X
l0 : list X
H : f_repeat x (S n') = x0 :: l0
IHn' : f_repeat x n' = x0 :: l0 -> r_repeat x n' (x0 :: l0)
=====
r_repeat x (S n') (x0 :: l0)
```

What are the next steps in the proof? What is the problem with this proof attempt after those steps have been taken? How might this problem be resolved? Be specific. (Use the next page if you need more space.)

(Extra space for the previous problem.)

## For Reference

```
Inductive nat : Type :=
  | 0 : nat
  | S : nat -> nat.
```

```
Inductive and (P Q : Prop) : Prop :=
  conj : P -> Q -> (and P Q).
```

```
Notation "P /\ Q" := (and P Q) : type_scope.
```

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

```
Inductive False : Prop := .
```

```
Definition not (P:Prop) := P -> False.
```

```
Notation "~ x" := (not x) : type_scope.
```

```
Notation "x <> y" := (~ (x = y)) : type_scope.
```

```
Fixpoint plus (n : nat) (m : nat) : nat :=
  match n with
  | 0 => m
  | S n' => S (plus n' m)
  end.
```

```
Notation "x + y" := (plus x y)(at level 50, left associativity) : nat_scope.
```

```
Fixpoint mult (n : nat) (m : nat) : nat :=
  match n with
  | 0 => 0
  | S n' => m + (mult n' m)
  end.
```

```
Notation "x * y" := (mult x y)(at level 40, left associativity) : nat_scope.
```

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

```

Fixpoint ble_nat (n m : nat) : bool :=
  match n with
  | 0 => true
  | S n' =>
    match m with
    | 0 => false
    | S m' => ble_nat n' m'
    end
  end.

```

```

Inductive beautiful : nat -> Prop :=
  b_0 : beautiful 0
| b_3 : beautiful 3
| b_5 : beautiful 5
| b_sum : forall n m, beautiful n -> beautiful m -> beautiful (n+m).

```

```

Inductive list (X:Type) : Type :=
  | nil : list X
  | cons : X -> list X -> list X.

```

```

Fixpoint app (X : Type) (l1 l2 : list X) : (list X) :=
  match l1 with
  | nil => l2
  | cons h t => cons X h (app X t l2)
  end.

```

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

```

Fixpoint map {X Y:Type} (f:X->Y) (l:list X) : (list Y) :=
  match l with
  | [] => []
  | h :: t => (f h) :: (map f t)
  end.

```

```

Fixpoint filter {X:Type} (test: X->bool) (l:list X) : (list X) :=
  match l with
  | [] => []
  | h :: t => if test h then h :: (filter test t)
              else filter test t
  end.

```