CIS 500 — Software Foundations

Midterm I

(Standard and advanced versions together)

September 30, 2014

Answer key

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

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

   Answer: True

   (b) The proof of an implication `P -> Q` is a function that uses a proof of the proposition `P` to produce a proof of the proposition `Q`.

   Answer: True

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

   Answer: False

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

   Answer: True

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

   Answer: False

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

   Answer: True

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

   Answer: False

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

   Answer: False

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

Answer: True

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

Answer: True

Grading scheme: 1 point each.

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`

Answer: `bool`

(b) `3=4`

Answer: `Prop`

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

Answer: `Prop`

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

Answer: `Prop -> Prop`

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

Answer: ill-typed

Grading scheme: 2 points for each correct type, and 0 points for wrong or missing type.

3. **[Standard]** (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`

Possible answers:

```
fun X (x:list X) => 0
fun X (x:list X) => length x
...
```

(b) `Prop`

Possible answers:

```
1 = 1
False
True
...
```

(c) `beautiful 8`
Possible answers:
`b_sum 5 3 b_5 b_3`
`b_sum 3 5 b_3 b_5`
...

(d) `forall (X:Prop), X -> ~~X`
Possible answers:
We know that `~~X = (X -> False) -> False` so one option is:
`fun X (x : X) (f : X -> False) => f x`

Grading scheme: 2 points for each correct expression, 1 point for partially correct expressions, and 0 points for wrong or missing expression.

4. **[Standard]** (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_l : forall n:nat, 0 * n = 0.`

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

Answer: iv

(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

Answer: ii

(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

Answer: iii

(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

Answer: ii

Grading scheme: 3 points for each correct answer.

5. [**Advanced**] (8 points) Recall the definition of `flat_map` from the homework (The `++` function is given in the references):

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

This function applies `f` to each element in the list and appends the results together. For example:

```
Example test_flat_map1:
  flat_map (fun (n:nat) => [n;n;n]) [1;5;4] = [1; 1; 1; 5; 5; 5; 4; 4; 4].
```

(a) Complete the definition of the list `filter` function using `flat_map`. (You will receive no credit if your answer uses `Fixpoint`!)

```
Definition filter {X : Type} (test: X->bool) (l:list X) : (list X) :=
```

Answer:

```
  flat_map (fun x => if (test x) then [x] else []) l.
```

Your `filter` should satisfy the same tests as the `filter` we saw in class. For example:

```
Example test_filter1: filter evenb [1;2;3;4] = [2;4].
```

(b) Complete the definition of the list `map` using `flat_map`. (You will receive no credit if your answer uses `Fixpoint`!)

```
Definition map {X Y:Type} (f : X -> Y) (l : list X) : list Y :=
```

Answer:

```
  flat_map (fun x => [f x]) l.
```

Again, your map should satisfy the same tests as the map we saw in class. For example:

```
Example test_map1: map (plus 3) [2;0;2] = [5;3;5].
```

Grading scheme: 4 points each. No deductions for minor syntax errors.

6. (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 :=`

One Answer: `j_app (j_one 3) j_nil`

`Definition example2 : jlist nat :=`

One Answer: `j_app j_nil (j_one 3)`

Grading scheme: 2 points total

```
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 :=
```

Answer:

```
  match x with
  | j_nil => j_nil
  | j_one y => j_one (f y)
  | j_app jl1 jl2 => j_app (j_map f jl1) (j_map f jl2)
  end.
```

Grading scheme: Five points total. -1 for too complex or otherwise minor problems. No deductions for minor syntax errors.

(c) What is the type of the expression `j_one` ? *Answer:* `forall (X : Type), X -> jlist X`
Grading scheme: 2 points

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

Answer: `jlist nat -> jlist bool`
Grading scheme: 2 points

(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 IHl1 l2 IHl2].
Case "j_nil".
   simpl. reflexivity.
Case "j_one".
   simpl. reflexivity.
Case "j_app".
   simpl. rewrite IHl1. rewrite IHl2. apply map_app.
Qed.
```

The `j_app` case of the `j_map` correctness proof makes use of two different induction hypotheses, called `IHl1` and `IHl2`. Circle the correct statement of `IHl1` used in this case of the proof.

i. `IHl1: to_list (j_app l1 l2) = map f (to_list (j_app l1 l2))`

ii. `IHl1: to_list (j_map f l1) = map f (to_list l1)`

iii. `IHl1: forall l1:jlist X. to_list (j_map f l1) = map f (to_list l1)`

iv. `IHl1: forall l2:jlist X. to_list (j_app l1 l2) = map f (to_list (j_app l1 l2))`

Answer: ii

Grading scheme: 3 points.

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).
```

Answer: iv

Grading scheme: 3 points.

7. [**Advanced**] (12 points) Write a *careful* informal proof of the following theorem. Make sure to state the induction hypothesis explicitly in the inductive step.

Theorem: Addition is commutative. For all x and y, $x + y = y + x$.

In your proof, you may use the following lemmas

- Lemma *plus_n_*0: 0 is a right identity for addition. i.e. for all n, $n + 0 = n$.
- Lemma *plus_n_Sm*: The successor of $(n + m)$ is equal to n plus the successor of m.

Answer: Let natural numbers n and m be given. We show $n + m = m + n$ by induction on m.

- First, suppose $m = 0$. We must show $n+0 = 0+n$. By the definition of $+$, we know $0+n = 0$, and we have already shown (lemma *plus_n_*0) that $n + 0 = 0$. Thus, showing $n + 0 = 0 + n$ is equivalent to showing $0 = 0$, which is true by reflexivity.
- Next, suppose $m = Sm'$ for some m', where $n+m' = m'+n$. We must show that $n+(Sm') = (Sm') + n$. By the definition of $+$ and the induction hypothesis, $(Sm') + n = S(m' + n) = S(n + m')$. It remains to show $n + (Sm') = S(n + m')$, which is precisely lemma *plus_n_Sm*.

Grading scheme:

- *2 pts for "induction on m"*
- *2 pts for having a base case*
- *2 pts for using plus_n_0 lemma correctly*
- *2 pts for having a successor case*
- *2 pts for using plus_n_Sm lemma correctly*
- *2 pts for using induction hypothesis correctly*
- *-1 for "not informal enough" English*

- *-1 for other minor errors*

8. (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 :=
```

Answer:

```
match n with
 | 0 => []
 | S n => x :: f_repeat x n
 end.
```

Grading scheme: 4 points

(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 :=
```

Answer:

```
| r_nil : forall x, r_repeat x O []
| r_cons : forall x n l, r_repeat x n l -> r_repeat x (S n) (x :: l).
```

Grading scheme: 4 points

(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.)

Answer: By simplification in `H` we know that `x :: f_repeat x n' = x0 :: l0`. By inversion, this gives us `x = x0` and `f_repeat x n' = l0`. Furthermore, by applying `r_cons`, we can reduce the goal to `r_repeat x l0`. But here we are stuck because the induction hypothesis only applies to `l`, not to `l0`.

We can fix this proof by not introducing `l` and `H` before the use of induction on `n`. If we do so, then our induction hypothesis will read

```
IHn' : forall l, f_repeat x n' = l -> r_repeat x n' l
```

Thus, we can apply it to the goal to complete the proof.

Grading Scheme: Five points total.

- observing that we can simplify and invert `H`
- observing that we should apply `r_cons` to reduce the goal
- observing that the induction hypothesis cannot be applied, even after we reduce `IHn'`
- proposing that we strengthen the induction hypothesis by not introducing `l` and `H` before the use of the induction tactic (or by using generalize dependent.)

(Extra space for the previous problem.)

For Reference

```
Inductive nat : Type :=
  | O : 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
    | O => 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
  | O, O => true
  | S n', S m' => beq_nat n' m'
  | _, _ => false
  end.
```

```
Fixpoint ble_nat (n m : nat) : bool :=
  match n with
  | O => true
  | S n' =>
      match m with
      | O => 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.
```