CIS 500 — Software Foundations

Midterm I

(Standard version)

February 13, 2013

Name: _____

Pennkey: _____

Scores:

1	
2	
3	
4	
5	
6	
7	
8	
Total (80 max)	

1. (12 points) Write the type of each of the following Coq expressions, or write "ill-typed" if it does not have one.

(a) `fun n:nat => [n]::nil`

(b) `forall X (l1 l2 : list X), l1 :: l2 = l2 :: l1`

(c) `forall X, X -> list X -> list X`

(d) `and False`

(e) `fun n1 => if beq_nat n1 0 then ble_nat 5 else beq_nat n1`

(f) `fun n : nat => forall m:nat, ble_nat m n = true`

2. (12 points) For each of the types below, write a Coq expression that has that type.

(a) `nat -> bool -> nat`

(b) `forall X, X -> list X`

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

(d) `Prop`

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

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

3. (10 points) Suppose that we wanted to prove the following theorem:

```
Theorem beq_nat_true : forall m n : nat, beq_nat m n = true -> m = n.
```

(The definition of `beq_nat` is given in the appendix, for easy reference.)

(a) What induction hypothesis will be generated by `induction` for the second subgoal (the induction step) if we start this way (doing `intros` on both `m` and `n`)?

```
Proof.  intros m n. induction m as [|m'].
```

(b) What induction hypothesis will be generated by `induction` for the second subgoal (doing `intros` on just `m`)?

```
Proof.  intros m. induction m as [|m'].
```

(c) Which of these two strategies is more likely to succeed? Why?

4. (8 points) Briefly explain the difference between the `apply` and `rewrite` tactics. (3-4 sentences at the most.)

5. (6 points) For each of the given theorems, which set of tactics is needed to prove it? (If more than one of the sets of tactics will work, choose the smallest set.)

(a) `forall n m : nat, beq_nat m n = true -> beq_nat (S m) (S n) = true`

i. `intros`, `simpl`, `rewrite`, `reflexivity`, and `induction`

ii. `intros`, `simpl`, `rewrite`, and `reflexivity`

iii. `intros`, `rewrite`, and `reflexivity`

iv. `intros` and `reflexivity`

(b) `forall (B : Prop), exists (A : Prop), A -> B`

i. `intros`, `exists`, and `rewrite`

ii. `intros`, `exists`, and `apply`

iii. `intros` and `exists`

iv. `intros` and `rewrite`

(c) `forall n, n + 0 = 0 -> n = 0`

i. `intros`, `rewrite`, `induction`, and `inversion`

ii. `intros`, `rewrite`, and `reflexivity`

iii. `intros`, `destruct`, and `reflexivity`

iv. `intros`, `destruct`, `inversion` and `reflexivity`

6. (10 points) A "fold function" captures a very common computation pattern: iterating over a data structure while accumulating a result. For instance, here is how you can use the fold function on lists (which we called simply `fold` in the notes) to sum all the elements of a list:

```
fold plus 0 [1,2,3,4]
```

When evaluated, this expression simplifies to `10`.

We can also write fold functions for other kinds of data structures. For example, consider the following definition of binary trees:

```
Inductive tree (X : Type) :=
| leaf : tree X
| node : X -> tree X -> tree X -> tree X.
```

In this problem, we will write a `fold_tree` function to perform the same pattern of iteration as in the list case. The type of `fold_tree` will be:

```
fold_tree : forall X Y, (X -> Y -> Y -> Y) -> Y -> tree X -> Y
```

Here is an example application of `fold_tree`:

```
fold_tree 0
          (fun b n1 n2 => b + n1 + n2)
          (node 4
             (node 3 leaf leaf)
             (node 1 leaf leaf))
```

When evaluated, this expression simplifies to `8`.

Complete the definition of `fold_tree` below.

```
Fixpoint fold_tree {X Y} (f : X -> Y -> Y -> Y)
                         (y : Y) (t : tree X) : Y :=
```

7. (10 points) In the first and second homework assignments, we worked with *binary numbers* encoded as a Coq inductive type. Here is one version of that encoding:

```
Inductive bin : Type :=
  | BZ : bin
  | T2 : bin -> bin
  | T2P1 : bin -> bin.
```

For example, `T2P1 (T2P1 BZ))` represents the number 3, while `T2 (T2 (T2P1 BZ))` represents 4.

(a) Recall that the `incr` function takes a binary number and returns its successor. For example, `incr (T2P1 (T2P1 BZ) = T2 (T2 (T2P1 BZ))`.

Complete the following definition of `incr`:

```
Fixpoint incr (m:bin) : bin :=
```

(b) The `bin_to_nat` function takes a binary number and returns its `nat` (unary) representation. For example, `bin_to_nat (T2P1 (T2P1 BZ)) = S (S (S O))`.

Complete the following definition of `bin_to_nat`:

```
Fixpoint bin_to_nat (m:bin) : nat :=
```

8. (12 points) In this problem, your task is to find a short English summary of the meaning of a proposition defined in Coq. For example, if we gave you this definition...

```
Inductive D : nat -> nat -> Prop :=
  | D1 : forall n, D n 0
  | D2 : forall n m, (D n m) -> (D n (n + m)).
```

... your summary could be "`D m n` holds when `m` divides `n` with no remainder."

(a)
```
Inductive R (X : Type) : X -> list X -> Prop :=
  | R1 : forall x l, R x (x::l)
  | R2 : forall x y l, (R x l) -> (R x (y::l)).
```

`R X x l` holds when:

(b)
```
Inductive R (X : Type) : list X -> list X -> Prop :=
  | R1 : R [] []
  | R2 : forall x l1 l2, (R l1 l2) -> (R l1 (x::l2))
  | R3 : forall x l1 l2, (R l1 l2) -> (R (x::l1) (x::l2)).
```

`R X l1 l2` holds when:

(c)
```
Inductive R (X : Type) : list X -> list X -> Prop :=
  | R1 : R [] []
  | R2 : forall x l1 l2 l3 l4,
           (R (l1++l2) (l3++l4)) ->
           (R (l1++[x]++l2) (l3++[x]++l4)).
```

`R X l1 l2` holds when:

(d)
```
Definition R (m : nat) :=
  m > 1 /\ (forall n, 1 < n -> n < m -> ~(D n m)).
```

(where `D` is given at the top of the page).

`R m` holds when:

For Reference

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


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


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

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


Inductive or (P Q : Prop) : Prop :=
  | or_introl : P -> or P Q
  | or_intror : Q -> or P Q.

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


Inductive False : Prop := .

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

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


Inductive ex (X:Type) (P : X->Prop) : Prop :=
  ex_intro : forall (witness:X), P witness -> ex X P.

Notation "'exists' x , p" := (ex _ (fun x => p))
  (at level 200, x ident, right associativity) : 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 beq_nat (n m : nat) : nat :=
  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.
```