CIS 500 — Software Foundations

Midterm I

(Standard version)

October 1, 2013

Name: ______________________________

Pennkey: ______________________________

Scores:

1	
2	
3	
4	
5	
6	
7	
Total (70 max)	

1. (12 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.)

(a) `fun n:nat => fun m:nat => n :: m :: n`

(b) `plus 3`

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

(d) `if beq_nat 0 1 then (fun n1 => beq_nat n1) else (fun n1 => ble_nat n1)`

(e) `forall (x:nat), beq_nat x x`

(f) `fun (X:Type) (x:X) => [x;x]`

2. (12 points) For each of the types below, write a Coq expression that has that type or write "Empty" if there are no such expressions.

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

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

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

(d) `nat -> Prop`

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

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

3. (7 points) Briefly explain the difference between `Prop` and `bool`. (3-4 sentences at the most.)

4. (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. (The definitions of `snoc` and `++` are given in the references.)

(a)
```
Lemma snoc_app : forall (X:Type) x (l1 l2:list X) ,
  (snoc l1 x) ++ l2 = l1 ++ (x::l2).
```

   i. `intros`, `simpl`, `rewrite`, and `reflexivity`
   ii. `intros`, `simpl`, `rewrite`, `reflexivity`, and `induction l1`
   iii. `intros`, `simpl`, `rewrite`, `reflexivity`, and `induction l2`
   iv. `intros`, `rewrite`, and `reflexivity`
   v. `intros` and `reflexivity`

(b) `forall (X:Type) (x y:X), snoc [] x = [y] -> x = y`

   i. `intros`, `inversion`, and `reflexivity`
   ii. `intros`, `destruct`, and `reflexivity`
   iii. `intros`, `destruct`, `inversion` and `reflexivity`
   iv. `intros`, `rewrite`, `induction`, and `inversion`

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

   i. `intros`, `exists`, and `rewrite`
   ii. `intros`, `exists`, and `apply`
   iii. `intros`, `exists`, and `inversion`
   iv. `intros` and `inversion`

5. (9 points) Recall the definition of `fold` from the homework:

```
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.
```

(a) Complete the definition of the list `length` function using `fold`.

```
Definition fold_length {X : Type} (l : list X) : nat :=
```

(b) Complete the definition of the list `map` function using `fold`.

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

(c) Complete the definition of the list `snoc` function using `fold`.

```
Definition fold_snoc {X:Type} (l:list X) v :=
```

6. (12 points) An alternate way to encode lists in Coq is the `dlist` ("doubly-ended list") type, which has a third constructor corresponding to the `snoc` operation on regular lists, as shown below:

```
Inductive dlist (X:Type) : Type :=
| d_nil : dlist X
| d_cons : X -> dlist X -> dlist X
| d_snoc : dlist X -> X -> dlist X.

(* Make the type parameter implicit. *)
Arguments d_nil {X}.
Arguments d_cons {X} _ _.
Arguments d_snoc {X} _ _.
```

We can convert any `dlist` to a regular `list` by using the following function (the definition of `snoc` on lists is given in the references).

```
Fixpoint to_list {X} (dl: dlist X) : list X :=
match dl with
| d_nil => []
| d_cons x l => x::(to_list l)
| d_snoc l x => snoc (to_list l) x
end.
```

(a) Just as we saw in the homework with the alternate "binary" encoding of natural numbers, there may be multiple `dlist`s that represent the same `list`. Demonstrate this by giving definitions of `example1` and `example2` such that the subsequent Lemma is provable (there is no need to prove it).

```
Definition example1 : dlist nat :=
```

```
Definition example2 : dlist nat :=
```

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

(b) It is also possible to define most list operations directly on the `dlist` representation. Complete the following function for appending two `dlist`s:

```
Fixpoint dapp {X} (l1 l2: dlist X) : dlist X :=
```

(c) The `dapp` function from part (b) should satisfy the following correctness lemma that states that it agrees with the list append operation. (The `++` function is given in the references.)

```
Lemma dapp_correct : forall (X:Type) (l1 l2:dlist X),
  to_list (dapp l1 l2) = (to_list l1) ++ (to_list l2).
Proof.
  intros X l1.
  induction l1 as [| x l| l x].
  Case "d_nil".
    ...
  Case "d_cons".
    ...
  Case "d_snoc".
    ...
Qed.
```

- What induction hypothesis is available in the `d_cons` case of the proof?

  i. `to_list (dapp (d_cons x l) l2) = (to_list (d_cons x l)) ++ (to_list l2)`

  ii. `to_list (dapp l l2) = (to_list l) ++ (to_list l2)`

  iii. `forall l2 : dlist X, to_list (dapp l l2) = to_list l ++ to_list l2`

  iv. `forall l2 : dlist X,`
  `to_list (dapp (d_cons x l) l2) = to_list (d_cons x l) ++ to_list l2`

- What induction hypothesis is available in the `d_snoc` case of the proof?

    i. `to_list (dapp (d_snoc x l) l2) = (to_list (d_snoc x l)) ++ (to_list l2)`

    ii. `to_list (dapp l l2) = (to_list l) ++ (to_list l2)`

    iii. `forall l2 : dlist X, to_list (dapp l l2) = to_list l ++ to_list l2`

    iv.
    ```
    forall l2 : dlist X,
      to_list (dapp (d_snoc x l) l2) = to_list (d_snoc x l) ++ to_list l2
    ```

7. (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) `Definition R (m : nat) := ~(D 2 m).`

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

`R m` holds when:

(b)
```
Inductive R {X:Type} : list X -> list X -> Prop :=
| R1 : forall l1 l2, R l1 (l1 ++ l2)
| R2 : forall l1 l2 x,  R l1 l2 -> R l1 (x::l2)
```

`R X l1 l2` holds when:

(c)
```
Inductive R {X:Type} (P:X -> Prop) : list X -> Prop :=
| R1 : R P []
| R2 : forall x l, P x -> R P l -> R P (x::l).
```

`R X P l` holds when:

(d)
```
Inductive R {X:Type} (P:X -> Prop) : list X -> Prop :=
| R1 : forall x l, P x -> R P (x::l)
| R2 : forall x l, R P l -> R P (x::l).
```

`R X P l` holds when:

For Reference

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


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


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


Fixpoint length (X:Type) (l:list X) : nat :=
  match l with
  | nil      => 0
  | cons h t => S (length X t)
  end.


Fixpoint index {X : Type} (n : nat)
               (l : list X) : option X :=
  match l with
  | [] => None
  | a :: l' => if beq_nat n O then Some a else index (pred n) l'
  end.


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 snoc (X:Type) (l:list X) (v:X) : (list X) :=
  match l with
  | nil      => cons X v (nil X)
  | cons h t => cons X h (snoc X t v)
  end.

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) : 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.
```