CIS 500 — Software Foundations

Midterm I

(Advanced version)

February 13, 2013

Name: ______________________________

Pennkey: ______________________________

Scores:

1	
2	
3	
4	
5	
6	
7	
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. (12 points) Write a *careful* informal proof of the following theorem. Make sure to state the induction hypothesis explicitly in the inductive step.

Theorem: `beq_nat m n = beq_nat n m`, for all natural numbers `m` and `n`.

5. (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 :=
```

6. (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:

7. (12 points) *Regular expressions* are a convenient way of describing sets of strings. Here's a definition of regular expressions (where the "characters" in the strings are numbers) as a Coq data type.

```
Inductive regex : Type :=
  | Literal : list nat -> regex
  | Union   : regex -> regex -> regex
  | Concat  : regex -> regex -> regex
  | Star    : regex -> regex.
```

Informally, a regular expression `r` matches a list of numbers `s` according to the following rules:

- `Literal s` only matches `s` itself.
- `Union r1 r2` matches strings that are matched by either `r1` or `r2`.
- `Concat r1 r2` matches by strings of the form `s1 ++ s2`, such that `s1` is matched by `r1` and `s2` is matched by `r2`.
- `Star r` matches a string `s` if `s = []` or if `s = s1 ++ s2 ++ ... ++ sn` and each substring is matched by `r`.

Your task (on the next page) will be to formalize the above specification by translating it into an inductive relation `matches` of type `regex -> list nat -> Prop`. For instance, the following propositions should be provable...

```
matches (Literal [1,2,3])                          [1,2,3]

matches (Union (Literal []) (Literal [2,1]))       []

matches (Concat (Literal [1,2,3]) (Literal [1]))   [1,2,3,1]

matches (Star (Literal [1]))                       [1,1,1,1,1]
```

...whereas the following shouldn't hold:

```
matches (Literal [1])                              [1,2,3]

matches (Star (Literal [2]))                       [1]

matches (Concat (Literal [3]) (Literal [3]))       [3,3,3]
```

Complete the definition of `matches` below.

```
Inductive matches : list nat -> regex -> Prop :=
```

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.
```