# CIS 500 — Software Foundations
# Midterm I

## Answer key

### February 18, 2009

1. (5 points) Consider the following Coq function:

```
 Fixpoint concatMap (X Y : Set) (f : X → list Y) (l : list X)
                    {struct l} : list Y :=
   match l with
   | nil    => nil
   | h :: t => (f h) ++ (concatMap _ _ f t)
   end.
```

   (a) What is the type of `concatMap`? (I.e., what does `Check concatMap` print?)

   *Answer:* `concatMap : forall X Y : Set, (X → list Y) → list X → list Y`

   (b) What does

   ```
   Eval simpl in (concatMap _ _ (fun x => x) [[1,2],[3,4]]).
   ```

   print?

   *Answer:* `= [1, 2, 3, 4] : list nat`

   (c) What does

   ```
   Eval simpl in (concatMap _ _ (fun x => [x+1,x+2]) ([1,2])).
   ```

   print?

   *Answer:* `= [2, 3, 3, 4] : list nat`

   *Grading scheme: -2 for each incorrect part*

2. (5 points)

   (a) Fill in the definition of the Coq function `elem` below.

   Given a type `X`, an equality-testing function `eq` for `X`, an element `e` of type `X`, and a list `l` of type `list X`, the expression `elem X eq e l` returns `true` if and only an element `eq`-equal to `e` appears in the list. For example, `elem nat beq_nat 2 [1,2,3]` yields `true` (because `beq_nat 2 2 = true`) while `elem nat beq_nat 5 [1,2,3]` yields `false`.

   ```
   Fixpoint elem (X : Set) (eq : X → X → bool) (e : X) (l : list X)
                 {struct l} : bool :=
   ```

   *Answer:*

   ```
   match l with
   | nil    => false
   | h :: t => orb (eq e h) (elem _ eq e t)
   end.
   ```

   *Grading scheme: -1 or -2 for various errors (hardly anyone missed this)*

(b) Why do we need to pass an equality-testing function `eq` as an argument to `elem` instead of just using = to test for equality?

*Answer: = yields a proposition, not a boolean*

*Grading scheme: -2 for failing to say something close to "= yields a proposition, not a boolean". Note that "because = is not polymorphic" (or words to that effect) is incorrect: = is polymorphic.*

3. (6 points) Fill in the definition of the Coq function `nub` below.

Given a type `X`, an equality function `eq` for `X`, and a list `l` of type `list X`, the expression `nub X eq l` yields a list that retains only the last copy of each element in the input list. For example, `nub nat beq_nat [1,2,1,3,2,2,4]` yields `[1,3,2,4]`.

```
Fixpoint nub (X : Set) (eq : X → X → bool) (l : list X)
             {struct l} : list X :=
```

*Answer:*

```
match l with
| nil     => nil
| h :: t  => if elem _ eq h t then nub _ eq t
                                else h :: (nub _ eq t)
end.
```

*Grading scheme: -1 to -5 for various errors (few people missed this question).*

4. (5 points)

(a) Briefly explain the use and behavior of the `apply` tactic.

*Answer: The* `apply` *tactic is used with a hypothesis from the current context or a previously defined theorem. If the conclusion of that hypothesis or theorem matches the current goal, it is eliminated and new subgoals are generated for each premise of the applied theorem. In this way,* `apply` *facilitates "backwards" reasoning.*

(b) Briefly explain the use and behavior of the `apply ... in ...` tactic.

*Answer:* `apply H1 in H2` *may be used when* `H2` *is a hypothesis in the current context.* `H1` *should be another hypothesis or a previously defined theorem, and a premise of* `H1` *must match* `H2`*. Using the tactic transforms* `H2` *into the conclusion of* `H1`*, and new subgoals are generated for each additional premise of* `H1`*.* `apply ... in ...` *facilitates forward reasoning.*

*Grading scheme: There was significant variation in this problem. Many errors other than the ones mentioned here are individually indicated. Common errors include: -1 point for not being general enough (suggesting* `apply`*/*`apply in` *only work when the applied hypothesis has exactly one premise); -2 points for saying that* `apply H1 in H2` *provides n new assumptions where* `H1` *has the form* `H2 → P1 → P2 ... → Pn`*; -1 point for saying* `apply...in...` *can be used to modify both assumptions and the goal.*

5. (6 points) Recall the Coq function `repeat`:

```
Fixpoint repeat (X : Set) (n : X) (count : nat) {struct count} : list X :=
  match count with
  | O => nil
  | S count' => cons n (repeat _ n count')
  end.
```

Consider the following partial proof:

```
Lemma repeat_injective : forall (X : Set) (x : X) (n m : nat),
  repeat _ x n = repeat _ x m →
  n = m.
Proof.
  intros X x n m eq. induction n as [|n'].
  Case "n = 0". destruct m as [|m'].
    SCase "m = 0". reflexivity.
    SCase "m = S m'". inversion eq.
  Case "n = S n'". destruct m as [|m'].
    SCase "m = 0". inversion eq.
    SCase "m = S m'".
      assert (n' = m') as H.
        SSCase "Proof of assertion".
```

Here is what the "goals" display looks like after Coq has processed this much of the proof:

```
2 subgoals

  SSCase := "Proof of assertion" : String.string
  SCase := "m = S m'" : String.string
  Case := "n = S n'" : String.string
  X : Set
  x : X
  n' : nat
  m' : nat
  eq : repeat X x (S n') = repeat X x (S m')
  IHn' : repeat X x n' = repeat X x (S m') → n' = S m'
  ============================
   n' = m'

subgoal 2 is:
 S n' = S m'
```

This proof attempt is not going to succeed. Briefly explain why and say how it can be fixed. (Do not write the repaired proof in detail—just say briefly what needs to be changed to make it work.)

*Answer: Because the induction hypothesis is insufficiently general. It gives us a fact involving one* particular `m`*, but to finish the last step of the proof we need to know something about a different* `m`*. To fix it, either use* `generalize dependent m` *before induction or do not intros* `m` *and* `eq` *to begin with.*

*Grading scheme: 3 points for identifying the problem, and 3 for explaining out to fix it. -2 points for not mentioning that the problem involves the IH. -1 point for being vague about nature of the problem (at the least, it should be made clear that the IH is too specific).*

6. (5 points) Suppose we make the following inductive definition:

```
Inductive foo (X : Set) (Y : Set) : Set :=
  | foo1 : X → foo X Y
  | foo2 : Y → foo X Y
  | foo3 : foo X Y → foo X Y.
```

Fill in the blanks to complete the induction principle that will be generated by Coq.

```
foo_ind
     : forall (X Y : Set) (P : foo X Y → Prop),

       (forall x : X, __________________________________) →
```

```
(forall y : Y, ___________________________________) →

(_________________________________________________) →

 _________________________________________________
```

*Answer:*

```
foo_ind
     : forall (X Y : Set) (P : foo X Y → Prop),
       (forall  x : X, P (foo1 X Y x)) →
       (forall  y : Y, P (foo2 X Y y)) →
       (forall f1 : foo X Y, P f1 → P (foo3 X Y f1)) →
       forall f2 : foo X Y, P f2
```

*Grading scheme: -1 point for missing forgetting the type arguments to foo's constructors. -2 points per line for other significant errors.*

7. (6 points)

Consider the following induction principle:

```
bar_ind
     : forall P : bar → Prop,
       (forall n : nat, P (bar1 n)) →
       (forall b : bar, P b → P (bar2 b)) →
       (forall (b : bool) (b0 : bar), P b0 → P (bar3 b b0)) →
       forall b : bar, P b
```

Write out the corresponding inductive set definition.

```
Inductive bar : Set :=

  | bar1 : ________________________________________

  | bar2 : ________________________________________

  | bar3 : ________________________________________.
```

*Answer:*

```
Inductive bar : Set :=
  | bar1 : nat → bar
  | bar2 : bar → bar
  | bar3 : bool → bar → bar.
```

*Grading scheme: Binary grading, 2pts per part.*

8. (6 points) Suppose we give Coq the following definition:

```
Inductive R : nat → list nat → Prop :=
  | c1 : R 0 []
  | c2 : forall n l, R n l → R (S n) (n :: l)
  | c3 : forall n l, R (S n) l → R n l.
```

Which of the following propositions are provable? (Write *yes* or *no* next to each one.)

(a) `R 2 [1,0]`     *Answer: Yes*

(b) `R 1 [1,2,1,0]` *Answer: Yes*

(c) `R 6 [3,2,1,0]` *Answer: No*

*Grading scheme: Binary grading, 2pts per part.*

9. (6 points) The following inductively defined proposition...

```
Inductive appears_in (X:Set) (a:X) : list X → Prop :=
  | ai_here : forall l, appears_in X a (a::l)
  | ai_later : forall b l, appears_in X a l → appears_in X a (b::l).
```

...gives us a precise way of saying that a value `a` appears at least once as a member of a list `l`.

Use `appears_in` to complete the following definition of the proposition `no_repeats X l`, which should be provable exactly when `l` is a list (with elements of type `X`) where every member is different from every other. For example, `no_repeats nat [1,2,3,4]` and `no_repeats bool []` should be provable, while `no_repeats nat [1,2,1]` and `no_repeats bool [true,true]` should not be.

```
Inductive no_repeats (X:Set) : list X → Prop :=
```

*Answer:*

```
| nr_nil : no_repeats X nil

| nr_cons : forall a l,
               no_repeats X l
             → ~ (appears_in X a l)
             → no_repeats X (a::l) .
```

*Grading scheme: Credit is was split 2pts for the nil case, 4pts for the cons case. Common deductions: -2 for forgetting to negate the* `appears_in` *predicate, -2 for forgetting inductive occurrence of* `no_repeats` *in the cons case, -1 for -4 ill-formed solutions.*

10. (2 points) Complete the definition of `and`, as it is defined in `Logic.v`:

```
Inductive and (A B : Prop) : Prop :=
```

*Answer:*

```
conj : A → B → (and A B).
```

*Grading scheme: 1 point for the constructor, 1 point for its type*

11. (2 points) Complete the definition of `or`, as it is defined in `Logic.v`:

```
Inductive or (A B : Prop) : Prop :=
```

*Answer:*

```
| or_introl : A → or A B
| or_intror : B → or A B.
```

*Grading scheme: 1 point for the constructors, 1 point for their types*

12. (6 points) Write an informal proof (in English) of the proposition ∀ `P : Prop`, `~(P ∧ ~P)`.

*Answer:*

Suppose, for some `P`, that `(P ∧ ~P)` holds. Recall that `~P` is defined as `P → False`. Given `P` and `P → False`, we can prove `False`, i.e. `(P ∧ ~P) → False`, i.e., `~(P ∧ ~P)`. *Grading scheme: +1 point for demonstrating knowledge of the definition of ~ or correctly beginning a proof by contradiction. +3 for correct proof outline. +1 point being clear about assumptions. +1 point for being clear about the nature of the contradiction. -1 or -2 points for bad style / sounding too much like coq.*

13. (4 points) Recall the `nat`-indexed proposition `ev` from `Logic.v`:

```
Inductive ev : nat → Prop :=
  | ev_0 : ev O
  | ev_SS : forall n:nat, ev n → ev (S (S n)).
```

Complete the definition of the following proof object:

```
Definition ev_plus2 : forall n, ev n → ev (plus 2 n) :=
```

*Answer:*

```
fun (n : nat) =>
   fun (E : ev n) =>
     ev_SS n E.
```

*Grading scheme: 1 point for* `ev_SS`*, 1 point for each function, 1 point for correct application of* `ev_SS`

14. (6 points) Recall the definition of `ex` (existential quantification) from `Logic.v`:

```
Inductive ex (X : Set) (P : X → Prop) : Prop :=
  ex_intro : forall witness:X, P witness → ex X P.
```

(a) In English, what does the proposition

```
ex nat (fun n => ev (S n))
```

mean?

*Answer: There is some number whose successor is even.*

*Grading scheme: 3 points for correct answer, -1 for imprecise language, -1 for claims about "all* `nat`*s"*

(b) Complete the definition of the following proof object:

```
Definition p : ex nat (fun n => ev (S n)) :=
```

*Answer:*

```
ex_intro nat (fun n => ev (S n)) 1 (ev_SS 0 ev_0).
```

*Grading scheme: 1 point for* `ex_intro`*, 1 point for the proof witness* `ev_SS 0 ev_0` *(or something equivalent, 1 point for correct application of* `ex_intro`

15. (10 points) Recall the definition of the `index` function:

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

Write an informal proof of the following theorem:

$$\forall \texttt{ X n l, length l = n} \rightarrow \texttt{index X (S n) l = None}.$$

*Answer:*

Let a set `X` and a list `l` be given. We will show, by induction on `l`, that `length l = n` implies `index X (S n) l = None`, for any natural number `n`. There are two cases to consider:

(a) If `l = nil`, we must show `index (S n) [] = None`. This follows immediately from the definition of `index`.

(b) Otherwise, `l = cons x :: l′` for some `x` and `l′`, where the induction hypothesis tells us that `length l′ = n′ => index (S n′) l = None` for any `n′`.

Let `n` be a number such that `length l = n`. We must show `index (S n) (x :: l′) = None`. By the definition of `index`, it is enough to show `index n l′ = None`.

But we know that `n = length l = length (x :: l′) = S (length l′)`. So it's enough to show `index (S (length l′)) l′ = None`, which follows directly from the induction hypothesis, picking `length l′` for `n′`.

*Grading scheme: 6 points for general proof structure. Common Errors: -1 minor confusion about variables or quantification, -2 for style problems, -1 not explaning that* `n` *has form* `S n′` *in the inductive step.*