CIS 500 — Software Foundations
Midterm I

Answer key

February 17, 2010

1. (10 points)

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

   The function is intended to be applied to a type `X`, an equality function `eq` for `X`, an element `x` of type `X`, and a list `l` of type `list X`. If if `l` already contains `x`, the returned list should be identical to `l`. On the other hand, if `l` does not already contain `x`, then the result should be a list which is identical to `l` except that it also contains `x` at the very end. For example, `insertUnique nat beq_nat 6 [1,2,4,3]` yields `[1,2,4,3,6]`, while `insertUnique nat beq_nat 4 [1,2,4,3]` yields `[1,2,4,3]`.

```
Fixpoint insertUnique (X : Type) (eq : X → X → bool) (x : X) (l : list X)
             : list X :=
```

   Answer:

```
match l with
| nil     => [x]
| h :: t  => if eq x h then h :: t
                       else h :: (insertUnique _ eq x t)
end.
```

   (b) Why do we need to pass an equality-testing function `eq` as an argument to `insertUnique` 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.*

2. (8 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.*

3. (10 points)

   Consider the following induction principle:

```
bar_ind
     : forall (X : Type) (P : bar X → Prop),
       P (bar1 X) →
       (forall (x : X) (f : bar X), P f → P (bar2 X x f)) →
       (forall (n : nat) (f : bar X), P f → P (bar3 X n f)) →
       forall (f : bar X), P f
```

   Write out the corresponding inductive type definition.

   Answer:

```
Inductive bar (X : Type) : Type :=
  | bar1 : bar X
  | bar2 : X → bar X → bar X
  | bar3 : nat → bar X → bar X.
```

   Grading scheme: Binary grading, 2pts per part.

4. (10 points) Suppose we make the following inductive definition:

```
 Inductive foo (X : Type) : Type :=
   | foo1 : foo X
   | foo2 : X → foo X
   | foo3 : nat → foo X → foo X.
```

   Write out the induction principle that will be generated by Coq.

   Answer:

```
foo_ind :
        forall (X : Type) (P : foo X → Prop),
        P (foo1 X) →
        (forall x : X, P (foo2 X x)) →
        (forall (n : nat) (f : foo X), P f1 → P (foo3 X n f)) →
        forall f : foo X, P f
```

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

5. (10 points) Define an inductive predicate `all_same X l`, which should be provable exactly when `l` is a list (with elements of type `X`) where all the elements are the same. For example, `all_same nat [1,1,1]` and `all_same nat []` should be provable, while `all_same nat [1,2,1]` and `all_same bool [true,false]` should not be.

```
Inductive all_same (X:Type) : list X → Prop :=
```

   Answer:

```
  | as_nil  : all_same X nil

  | as_sing : forall (x : X), all_same X [x]

  | as_cons : forall x l,
                all_same X (x :: l) →
                all_same X (x :: x :: l).
```

6. (8 points) Recall the `appears_in` relation, which expresses that an element `a` appears in a list `l`.

```
Inductive appears_in (X:Type) (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).
```

Complete the definition of the following proof object:

```
Definition appears_example : forall x y : nat, appears_in nat 4 [x,4,y] :=
```

Answer:

```
fun (x y : nat) => ai_later nat 4 x [4,y] (ai_here nat 4 [y]).
```

7. (8 points)

Consider the following partial proof:

```
Theorem toil_and_trouble : forall n m,
     double n = double m →
     n = m.
Proof.
  intros n m. induction n as [| n'].
  Case "n = O". simpl. intros eq. destruct m as [| m'].
    SCase "m = O". reflexivity.
    SCase "m = S m'". inversion eq.
  Case "n = S n'". intros eq. destruct m as [| m'].
    SCase "m = O". inversion eq.
    SCase "m = S m'".
      assert (n' = m') as H.
      SSCase "Proof of assertion".
        (* stopped here *)
```

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
  n' : nat
  m' : nat
  IHn' : double n' = double (S m') → n' = S m'
  eq : double (S n') = double (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).

8. (16 points) Give an informal proof, in English, of the following theorem.

```
Theorem distr_rev : forall X:Type, forall l1 l2 : list X,
  rev (l1 ++ l2) = (rev l2) ++ (rev l1).
```

The definition of `rev` is given in the exam Appendix. You may assume without proof the two lemmas `app_nil_end` and `snoc_with_append`, which are also stated in the Appendix.

Proof: Consider an arbitrary type `X` and an arbitrary list `l2`. We show that, for all `l1 : list X`, `rev (l1 ++ l2) = (rev l2) ++ (rev l1)`, by induction on `l1`.

- Suppose `l1 = []`. We must show that `rev ([] ++ l2) = rev (l2 ++ [])`. The left hand side of the equality simplifies to `rev l2`. By Lemma `app_nil_end` the right hand side is also equal to `rev l2`.
- Suppose `l1 = x::l1′`, where `rev (l1′++l2) = (rev l2) ++ (rev l1′)`. We must show that `rev (x::l1′ ++ l2) = rev l2 ++ rev (x::l1′)`. The left hand side of the equality simplifies to `snoc (rev (l1′++l2)) x` and the right hand side simplifies to `rev l2 ++ snoc (rev l1′) x`. Now by the induction hypothesis the left hand side is equal to `snoc ((rev l2) ++ (rev l1)) x`, which by Lemma `snoc_with_append` is equal to `rev l2 ++ snoc (rev l1′) x` as required.

Appendix

The function `rev` is defined as follows.

```
Fixpoint snoc {X:Type} (l:list X) (x:X) : list X :=
  match l with
  | nil    => [x]
  | h :: t => h :: (snoc t x)
  end.

Fixpoint rev {X:Type} (l:list X) : list X :=
  match l with
  | nil    => nil
  | h :: t => snoc (rev t) h
  end.
```

For question 8 you may assume without proof the following two lemmas.

```
Theorem app_nil_end : forall X (l : list X),
  l ++ [] = l.

Theorem snoc_with_append : forall X : Type,
                           forall l1 l2 : list X,
                           forall x : X,
  snoc (l1 ++ l2) x = l1 ++ (snoc l2 x).
```