CIS 500 — Software Foundations

Midterm I

Review questions with answers

October 10, 2007

Work each of the review problems yourself before looking at the answers given here. If your answer differs from ours, make sure you understand why.

Functional Programming

1. Consider the following `Fixpoint` definition in Coq.

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

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

   Answer: `fold : forall X Y : Set, (X -> Y -> Y) -> list X -> Y -> Y`

2. What does

```
Check fold _ _ plus.
```

   print?

   Answer: `fold nat nat plus : list nat -> nat -> nat`

3. What does

```
Eval simpl in (fold _ _ plus [one,two,three,four] one).
```

   print?

   Answer: `= eleven : nat`

4. Recall that the `map` function takes a function `f` of type `X->Y` and a list `l` with elements of `X` and returns a list with elements of type `Y` containing the result of applying `f` to each element of `l`.

   Which of the following `Fixpoint` definitions correctly implements the `map` function? Circle the correct answer.

   (a) `Fixpoint m1 (X:Set) (Y:Set) (f:X->Y) (l:list X) {struct l} : (list Y) := match l with | nil => nil Y | h :: t => h :: (m1 X Y f t) end.`

   (b) `Fixpoint m2 (X:Set) (Y:Set) (f:X->Y) (l:list X) {struct l} : (list Y) := match l with | nil => f (nil Y) | h :: t => (f h) :: (m2 X Y f t) end.`

   (c) `Fixpoint m3 (X:Set) (Y:Set) (f:X->Y) (l:list X) {struct l} : (list Y) := match l with | nil => nil Y | h :: t => (f h) :: (m3 X Y f t) end.`

   (d) `Fixpoint m4 (X:Set) (Y:Set) (f:X->Y) (l:list X) {struct l} : (list Y) := match l with | nil => nil Y | h :: t => f h :: f (m4 X Y f t) end.`

   (e) `Fixpoint m5 (X:Set) (Y:Set) (f:X->Y) (l:list X) {struct l} : (list Y) := match l with | nil => nil Y | h :: t => f h :: f t end.`

   (f) None of the above.

   Answer: c

5. The `every` function takes a predicate `p` (a one-argument function returning a `yesno`) and a list `l`; it returns `yes` if `p` returns `yes` on every element of `l` and `no` otherwise.

```
Lemma check_filter1:
  every nat even [two,four,zero] = yes.
Proof. simpl. reflexivity. Qed.

Lemma check_filter2:
  every nat even [two,one,four,zero] = no.
Proof. simpl. reflexivity. Qed.
```

What is the type of `every`?

Answer: `forall X : Set, (X -> yesno) -> list X -> yesno`

6. Complete the following definition of `every` as a recursive function:

```
Fixpoint every (X:Set) (p:X->yesno) (l:list X) {struct l} : yesno :=

  match l with

  | nil    => _____________________

  | h :: t => both_yes  ____________________________   _________________________________

  end.
```

Answer:

```
Fixpoint every (X:Set) (p:X->yesno) (l:list X) {struct l} : yesno :=
  match l with
  | nil    => yes
  | h :: t => both_yes (p h) (every _ p t)
  end.
```

7. Complete the following definition of `every` by supplying appropriate arguments to `fold`:

```
Definition every' (X:Set) (p:X->yesno) (l:list X) : yesno :=

  fold _ _

     (fun x acc => both_yes _______________  _____________) __________  __________.
```

Answer:

```
Definition every' (X:Set) (p:X->yesno) (l:list X) : yesno :=
  fold _ _ (fun x acc => both_yes (p x) acc) l yes.
```

Coq Basics

8. Briefly explain the difference between the tactics `apply` and `rewrite`. Are there situations where either one can be applied?

   Answer: The ***`rewrite`*** *tactic is used to apply a known equality to either the goal or a hypothesis in the context, replacing all occurrences of one side by the other. The* ***`apply`*** *tactic uses a known implication (a hypothesis from the context, a previously proved lemma, or a constructor) to modify the proof state either backward (if the goal matches the conclusion of the implication, in which case a subgoal is generated for each premise of the implication) or forward (if some hypothesis matches the premise of the implication, in which case this hypothesis is replaced by the conclusion of the implication). If the fact is itself an equality (i.e., an implication with no premises), then either tactic can be used.*

9. Briefly explain the difference between the tactics `destruct` and `induction`.

   Answer: Both are used to perform case analysis on an element of an inductively defined type; ***`induction`*** *also generates an induction hypothesis, while* ***`destruct`*** *does not.*

10. The following proof attempt is not going to succeed. Briefly explain why and how it can be fixed.

```
Lemma double_injective : forall m n,
     double m = double n
  -> m = n.
Proof.
  intros m n. induction m.
    Case "O". simpl. intros eq. destruct n.
      Case "O". reflexivity.
      Case "S". inversion eq.
    Case "S". intros eq. destruct n.
      Case "O". inversion eq.
      Case "S".
        assert (m = n) as H.
          Case "Proof of assertion".
            ...
```

   Answer: Because the induction hypothesis is insufficiently general — it gives us a fact involving one particular ***`n`****, but to finish the last step of the proof we need to know something about a different* ***`n`****. To fix it, either use* ***`generalize dependent n`*** *before* ***`induction`*** *or else just omit* ***`intros m n`****.*

Inductively Defined Sets

11. Suppose we give Coq the following declarations:

```
Inductive mumble : Set :=
  | a : mumble
  | b : mumble -> nat -> mumble
  | c : mumble.

Inductive grumble (X:Set) : Set :=
  | d : mumble -> grumble X
  | e : X -> grumble X.
```

    Which of the following are well-typed members of the set `grumble`?

    (a) `d (b a five)` *Answer: No*

    (b) `d mumble (b a five)` *Answer: Yes*

    (c) `d yesno (b a five)` *Answer: Yes*

    (d) `e yesno yes` *Answer: Yes*

    (e) `e mumble (b c zero)` *Answer: Yes*

    (f) `e yesno (b c zero)` *Answer: No*

    (g) `c` *Answer: No*

12. Consider the following inductive definition:

```
Inductive baz : Set :=
  | x : baz -> baz
  | y : baz -> yesno -> baz.
```

    How *many* elements does the set `baz` have?

    Answer: None.

13. Consider the following inductive definition:

```
Inductive tree : Set :=
  | leaf : tree
  | node : tree -> tree -> tree.
```

Describe, in English, the elements of the set `tree`.

Answer: Unlabeled binary trees.

14. What induction principle will Coq generate for `tree`?

Answer:

```
tree_ind
     : forall P : tree -> Prop,
            P leaf
         -> (forall t : tree,
                  P t
               -> forall t0 : tree,
                     P t0
                  -> P (node t t0))
         -> forall t : tree, P t
```

Alternate answer (the one above is what Coq actually generates, but it is logically equivalent to this arguably more natural variant):

```
tree_ind
     : forall P : tree -> Prop,
            P leaf
         -> (forall t1 t2 : tree,
                  P t1
               -> P t2
               -> P (node t1 t2))
         -> forall t : tree, P t
```

15. Here is an induction principle for an inductively defined set `s`.

```
myset_ind :
  forall P : myset -> Prop,
      (forall y : yesno, P (con1 y))
   -> (forall (n : nat) (m : myset), P m -> P (con2 n m))
   -> forall m : myset, P m
```

What is the definition of `myset`?

Answer:

```
Inductive myset : Set :=
  | con1 : yesno -> myset
  | con2 : nat -> myset -> myset.
```

16. Consider the following inductive definition:

```
Inductive stree : nat -> Set :=
  | leaf : stree one
  | node : forall m n, stree m -> stree n -> stree (plus m n).
```

Describe, in English, the elements of the set `stree seven`.

Answer: Unlabeled binary trees with exactly seven leaves. (In general, for each ***n****, the set* ***stree n*** *contains binary trees with* ***n*** *total leaves.)*

Inductively Defined Propositions

17. Consider the following inductively defined family of propositions:

```
Inductive p : tree -> nat -> Prop :=
  | c1 : p leaf one
  | c2 : forall t1 t2 n1 n2,
           p t1 n1 -> p t2 n2 -> p (node t1 t2) (plus n1 n2).
```

Describe, in English, the conditions under which the proposition `p t n` is provable.

Answer: This proposition is provable when `t` *is an unlabeled binary tree with exactly* `n` *leaves. (That is* `p` *is a relation between trees and numbers that relates each tree to its number of leaves.)*

18. Consider the following inductively defined family of propositions:

```
Inductive bar : nat -> Prop :=
  | d : bar six
  | e : forall n, bar (times n n)
  | f : bar three -> bar five.
```

For which `n` is the proposition `bar n` provable?

Answer: `bar n` *is provable when* `n` *is either* `six` *or a perfect square (*`zero`*,* `one`*,* `four`*,* `nine`*, etc.).*

"Programming with Propositions"

19. The concept of *composition of relations* can be defined as follows:

    > Suppose `Q` and `R` are both relations on a set `X`. The *composition* of `Q` and `R` is the relation `C` such that, for all `x` and `z` in `X`,
    >
    > $$\texttt{C x z} \quad \longleftrightarrow \quad \exists \texttt{y}.\ \texttt{Q x y} \wedge \texttt{R y z}.$$

    Write an `Inductive` definition in Coq that expresses this concept.

    Answer:

```
Inductive composition (X: Set) (Q R: relation X) : X -> X -> Prop :=
  comp : forall x z, (exists y, Q x y /\ R y z) -> composition X Q R x z.
```

20. Give the definition of logical conjunction (`and`) as an inductive proposition in Coq.

    Answer:

```
Inductive and (A B : Prop) : Prop :=
  conj : A -> B -> (and A B).
```

Operational Semantics

21. Recall the definition of the set `tm`, the predidate `value`, and the relation `eval` from the lecture notes:

```
Inductive tm : Set :=
  | tm_const : nat -> tm
  | tm_plus : tm -> tm -> tm.

Inductive value : tm -> Prop :=
  v_const : forall n, value (tm_const n).

Inductive eval : tm -> tm -> Prop :=
  | E_PlusConstConst : forall n1 n2,
        eval (tm_plus (tm_const n1) (tm_const n2))
             (tm_const (plus n1 n2))
  | E_Plus1 : forall t1 t1' t2,
        (eval t1 t1')
     -> eval (tm_plus t1 t2)
             (tm_plus t1' t2)
  | E_Plus2 : forall v1 t2 t2',
        (value v1)
     -> (eval t2 t2')
     -> eval (tm_plus v1 t2)
             (tm_plus v1 t2').
```

Which of the following propositions are provable?

(a)
```
eval (tm_plus (tm_const one) (tm_const two))
     (tm_const (plus one two)).
```
Answer: Yes

(b)
```
eval (tm_plus
        (tm_plus (tm_const one) (tm_const two))
        (tm_const three))
     (tm_const (plus (plus one two) three)).
```
Answer: No

(c)
```
eval (tm_plus
        (tm_plus (tm_const one) (tm_const two))
        (tm_plus (tm_const three) (tm_const four)))
     (tm_plus
        (tm_plus (tm_const one) (tm_const two))
        (tm_const (plus three four))).
```
Answer: No

Recall the determinism and progress theorems for the `eval` relation.

```
Theorem eval_deterministic :
  partial_function _ eval.

Theorem eval_progress : forall t,
  value t \/ (exists t', eval t t').
```

22. (a) Suppose we add a new constructor to the evaluation relation as follows:

```
| E_Funny1 :
      eval (tm_const zero)
           (tm_const zero)
```

   i. Does `eval_deterministic` continue to hold? *Answer: No*

   ii. Does `eval_progress` continue to hold? *Answer: Yes*

   (b) Suppose we remove the constructor `v_const` from the definition of `value`.

   i. Does `eval_deterministic` continue to hold? *Answer: Yes*

   ii. Does `eval_progress` continue to hold? *Answer: No*

   (c) Is there any way we can cause `eval_progress` to fail by *adding* new constructors to the definition of `eval`? *Answer: No*