CIS 500 — Software Foundations
Midterm I

Answer key

October 10, 2007
1. (2 points) Consider the following Fixpoint definition in Coq.

```coq
Fixpoint zip (X Y : Set) (lx : list X) (ly : list Y) {struct lx}
    : list (X*Y) :=
  match lx with
  | nil =>
      nil _
  | x::tx =>
      match ly with
      | nil => nil _
      | y::ty => (x,y) :: (zip _ _ tx ty)
    end
  end.
```

What is the type of \texttt{zip}? (I.e., what does \texttt{Check zip} print?)

\textit{Answer:} \texttt{forall X Y : Set, list X -> list Y -> list (X * Y)}

2. (2 points) What does

\[ \text{Eval simpl in (zip _ _ [one,two] [no,no,yes,yes]).} \]

\texttt{print?}

\textit{Answer:} \texttt{= [(one, no), (two, no)]}
3. (6 points) Intuitively, the `zip` function transforms a pair of lists into a list of pairs. The inverse transformation, `unzip`, takes a list of pairs and returns a pair of lists. For example, evaluating

```
Eval simpl in (unzip _ _ [(one,no),(two,no)]).
```

prints:

```
= ([one, two], [no, no])
: list nat * list yesno
```

Fill in the blanks in the following definition of `unzip`. Write out all type parameters explicitly — do not use the “wildcard type” anywhere (i.e., please be *completely explicit* about type parameters here, rather than leaving them implicit as we did in the definition of `zip` above).

```
Fixpoint unzip (X Y : Set) (l : list (X*Y)) {struct l}
: __________________________________________ :=

match l with
| nil => ________________________________________
| cons (x,y) t =>

  match unzip X Y t with ____________________________

  end

end.
```

*Answer:*

```
Fixpoint unzip (X Y : Set) (l : list (X*Y)) {struct l}
: (list X) * (list Y) :=

match l with
| nil =>

  (nil X, nil Y)
| cons (x,y) t =>

  match unzip X Y t with ____________________________

  (lx,ly) => (x::lx, y::ly)

end

end.
```

*Grading scheme: 1 point for the type, 2 for the `nil` case, 3 for the `cons` case. -1 for incorrect type, for minor errors in `nil` or `cons` case, or for not providing explicit types. -2 for using `nil` subcase in the `cons` case or half-correct `cons` case. -3 for `cons` case missing or incorrect.*
4. (3 points) Recall that the filter function takes a function test of type X->yesno and a list l with elements of X and returns a list with elements of type X containing just the elements of l for which test yields yes.

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

(a) Fixpoint filter (X:Set) (test: X->yesno) (l:list X) {struct l} : (list X) :=
    match l with
    | nil => nil X
    | h :: t => match (test h) with
    | no => h :: (filter X test t)
    | yes => filter X test t
    end
end.

(b) Fixpoint filter (X:Set) (test: X->yesno) (l:list X) {struct l} : (list X) :=
    match l with
    | nil => nil X
    | h :: t => match (test l) with
    | no => filter X test t
    | yes => h :: (filter X test t)
    end
end.

(c) Fixpoint filter (X:Set) (test: X->yesno) (l:list X) {struct l} : (list X) :=
    match l with
    | nil => nil X
    | h :: t => h :: (filter X test t)
end.

(d) Fixpoint filter (X:Set) (test: X->yesno) (l:list X) {struct l} : (list X) :=
    match l with
    | nil => nil X
    | h :: t => match (test h) with
    | no => filter X test t
    | yes => h :: (filter X test t)
    end
end.

(e) Fixpoint filter (X:Set) (test: X->yesno) (l:list X) {struct l} : (list X) :=
    match l with
    | nil => nil X
    | h :: t => (test h) :: (filter X test t)
end.

(f) None of the above.

Answer: d
5. (4 points) Briefly explain what the assert tactic does.

Answer: assert e as H causes Coq to replace the current goal G with two subgoals: (1) e and (2) G again, but this time with the assertion e as a hypothesis in the context (called H).

Grading scheme: Most answers to this one were at least partially correct, so full credit was given only for “very correct” answers. In particular, we deducted -1 for not mentioning, in some way, that the asserted proposition gets added to the context for the remainder of the proof. -1 for other small confusions. -2 for very short or more seriously confused explanations.

6. (5 points) Recall the definition of the inductive predicate evenI:

\[
\text{Inductive evenI : nat -> Prop :=}
\]
\[
\mid \text{even_zero : evenI 0}
\]
\[
\mid \text{even_SS : forall n:nat, evenI n -> evenI (S (S n))}.
\]

The following proof attempt is not going to succeed. Briefly explain why.

Lemma l : forall n, evenI n.
Proof.
  intros n. induction n.
  Case "O". simpl. apply even_zero.
  Case "S".
  ...

Answer: Because the claim being proven is false. Grading scheme: To receive any credit, an answer had to suggest, in some way, that the claim was false. Partial credit was given for explaining in what way the proof seemed stuck.

7. (6 points) Recall the definition of plus:

\[
\text{Fixpoint plus (m : nat) (n : nat) {struct m} : nat :=}
\]
\[
\text{match m with}
\]
\[
\mid O => n
\]
\[
\mid S m' => S (plus m' n)
\]
\[
\text{end.}
\]

Grading scheme: 5/5 - Explanation including fact that lemma is not true. 3-4/5 - Mostly clear, but insufficiently explicit (need not suggest that lemma is false). 1-2/5 - Potentially very unclear explanation, composed of true statements explaining local problems with the proof state. 0/5 - Incorrect explanation (automatic zero for suggesting lemma can be proved using a particular technique).

(a) What will Coq print in response to this query?

Eval simpl in (forall n, plus n 0 = n).

Answer: forall n : nat, plus n 0 = n

(b) What will Coq print in response to this query?

Eval simpl in (forall n, plus 0 n = n).

Answer: forall n : nat, n = n
(c) Briefly (1 or two sentences) explain the difference.

Answer: Since \texttt{plus} is defined by structural recursion on its first argument, the definition tells us immediately that \texttt{plus \ O \ n} is \texttt{n}, without having to know what \texttt{n} is. On the other hand, \texttt{plus \ n \ O} cannot be simplified without knowing whether \texttt{n} has the form \texttt{O} or \texttt{S \ #}.

Grading scheme: Two points for each part. One point off for omitting \texttt{forall \ n} from one or both of the first two parts.
8. (8 points) Consider the following inductive definition:

\[
\text{Inductive foo (X:Set) : Set :=}
| \ c1 : \text{list X} \to \text{foo X} \to \text{foo X}
| \ c2 : \text{foo X}.
\]

What induction principle will Coq generate for foo? (Fill in the blanks.)

foo_ind :

\[
\text{forall (X : Set) (P : foo X \to Prop),}
\]

\[
(\text{forall (l : list X) (f : foo X),}
\]

\[
\_________________________ \to \__________________________}
\]

\[
\__________________________}
\]

\[
\to \forall f : \text{foo X}, \__________________________
\]

Answer:

foo_ind :

\[
\text{forall (X : Set) (P : foo X \to Prop),}
\]

\[
(\text{forall (l : list X) (f : foo X),}
\]

\[
\text{P f \to P (c1 X l f))}
\]

\[
\text{P (c2 X)}
\]

\[
\forall f : \text{foo X}, \text{P f}
\]

Grading scheme: -2 points for not putting parentheses around the constructor applications, -3 points for omitting the X parameter from constructor applications, -1 to -3 points for each other mistake.

9. (6 points) Here is an induction principle for an inductively defined set s.

\[
\text{myset_ind :}
\]

\[
\text{forall P : myset \to Prop,}
\]

\[
(\text{forall y : yesno, P (con1 y))}
\]

\[
\to (\text{forall n : nat (m : myset), P m \to P (con2 n m))}
\]

\[
\to \forall m : \text{myset}, \text{P m}
\]

What is the definition of myset?

Answer:

\[
\text{Inductive myset : Set :=}
\]

\[
| \ \text{con1 : yesno \to myset}
\]

\[
| \ \text{con2 : nat \to myset \to myset.}
\]

Grading scheme: -1 for omitting the keyword "Set" in the definition. -2 for providing wrong type once.
Inductively Defined Propositions

10. (6 points) Recall the definition of the set \texttt{tree} from the review exercises:

\[
\begin{align*}
\text{Inductive tree : Set :=} \\
& \mid \text{leaf : tree} \\
& \mid \text{node : tree -> tree -> tree}.
\end{align*}
\]

Now consider the following inductively defined proposition:

\[
\begin{align*}
\text{Inductive p : tree -> nat -> Prop :=} \\
& \mid c1 : p \text{ leaf one} \\
& \mid c2 : \forall t1 t2 n1 n2, \\
& \hspace{1em} p t1 n1 \rightarrow p t2 n2 \rightarrow p (\text{node } t1 t2) (\text{plus } n1 n2) \\
& \mid c3 : \forall t n, p t n \rightarrow p t (S n).
\end{align*}
\]

Describe, in English, the conditions under which the proposition \( p \, t \, n \) is provable.

\textit{Answer: This proposition is provable when }\( t \)\textit{ is an unlabeled binary tree with at most }\( n \)\textit{ leaves. (That is }p\textit{ is a relation between trees and numbers that relates each tree }t\textit{ and number }n\textit{ such that the number of leaves in }t\textit{ is less than or equal to }n\textit{.)} \textit{Grading scheme: -1 point for writing “at least” instead of “at most.” -3 or -4 for getting the condition “not completely wrong” in other ways.}
11. (4 points) Suppose we give Coq the following definition:

```coq
Inductive R : nat -> nat -> nat -> Prop :=
  | c1 : R zero zero zero
  | c2 : forall m n o, R m n o -> R (S m) n (S o)
  | c3 : forall m n o, R m n o -> R m (S n) (S o)
  | c4 : forall m n o, R (S m) (S n) (S (S o)) -> R m n o
  | c5 : forall m n o, R m n o -> R n m o.
```

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

(a) R one one two Answer: Yes
(b) R two two six Answer: No

Grading scheme: Two points for each item.

12. (2 points) If we dropped constructor c5 from the definition of R, would the set of provable propositions change? Write yes or no and briefly (1 sentence) explain your answer. Write yes or no. Answer: No

Grading scheme: One point for answer, one point for text justification, which is that symmetry of n and m is derivable.

13. (2 points) If we dropped constructor c4 from the definition of R, would the set of provable propositions change? Write yes or no. Answer: No

Grading scheme: Two points for correct answer, zero for wrong answer.
“Programming with Propositions”

14. (5 points) Complete the following definition of existential quantification as an inductive proposition in Coq.

\[
\text{Inductive ex} (X : \text{Type}) (P : X \rightarrow \text{Prop}) : \text{Prop} :=
\]

\[
\text{ex_intro :} \quad \text{forall} \ \text{witness :} X, \ P \ \text{witness} \ \rightarrow \ \text{ex} X P.
\]

Answer:

\[
\text{ex_intro : forall witness :} X, \ P \ \text{witness} \ \rightarrow \ \text{ex} X P.
\]

Grading scheme: Approximately, one point for \textit{forall witness :} X, two points for \textit{P witness}, and two points for \textit{ex X P}. No credit taken off for omitting the constructor name. Any constructor name accepted, and, of course, any name accepted for the bound variable.
15. (6 points) In the lectures, we have been working with a very simple programming language whose terms consist of just `plus` and constants. Here is an equally simple language whose terms are just the boolean constants `true` and `false` and a conditional expression:

```
Inductive tm : Set :=
| tm_true : tm
| tm_false : tm
| tm_if : tm -> tm -> tm -> tm.
```

```
Inductive value : tm -> Prop :=
| v_true : value tm_true
| v_false : value tm_false.
```

```
Inductive eval : tm -> tm -> Prop :=
| E_IfTrue : forall t1 t2, eval (tm_if tm_true t1 t2) t1
| E_IfFalse : forall t1 t2, eval (tm_if tm_false t1 t2) t2
| E_If : forall t1 t1' t2 t3, eval t1 t1'
  -> eval (tm_if t1 t2 t3)
  (tm_if t1' t2 t3).
```

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

(a) `eval tm_false tm_false`
   
   Answer: No

(b) `eval (tm_if tm_true tm_true tm_true)
    (tm_if tm_true tm_true tm_true)
    (tm_if tm_false tm_false tm_false)) tm_true`

   Answer: No

(c) `eval (tm_if tm_true tm_true tm_true)
    (tm_if tm_true tm_true tm_true)
    tm_false)
    (tm_if tm_true tm_true tm_true)
    tm_false)`

   Answer: Yes
16. (4 points) Suppose we want to add a “short circuit” to the evaluation relation so that it can recognize when the then and else branches of a conditional are the same value (either tm_true or tm_false) and reduce the whole conditional to this value in a single step, even if the guard has not yet been reduced to a value. For example, we would like this proposition to be provable:

```plaintext
eval
  (tm_if
    (tm_if tm_true tm_true tm_true)
    tm_false
    tm_false)
  tm_false
```

Write an extra clause for the eval relation that achieves this effect. (Fill in the blanks.)

| E_ShortCircuit : forall ____________________________, ______________________________________________________|
| Answer: | E_ShortCircuit : forall t1 t23, (value t23) -> eval (tm_if t1 t23 t23) t23 |

Grading scheme: -2 if branches of conditional are not constrained to be equal. -2 for forgetting value t23 premise. -1 for using an eq constraint of form t2=t3 instead of letting unification handle this. Max score 1/4 for giving a rule that only works on the example, evaluating under branches or giving irrelevant premises in place of value _.
17. (9 points) It can be shown that the determinism and progress theorems for the eval relation in the lecture notes...

\[ \text{Theorem eval_deterministic :} \]
\[ \text{partial_function _ eval.} \]

\[ \text{Theorem eval_progress : } \forall t, \]
\[ \text{value t } \lor (\exists t', \text{ eval t t'}). \]

...also hold for the definition of eval given in question 15.

After we add the clause E_ShortCircuit from question 16...

(a) Does eval_deterministic still hold? Write yes or no and briefly (1 sentence) explain your answer. Answer: No. For example, the term in question 16 now evaluates to both \( \text{tm_false} \) and \( \text{tm_if tm_true tm_false tm_false} \).

(b) Does eval_progress still hold? Write yes or no and briefly (1 sentence) explain your answer. Answer: Yes – adding a new constructor to eval doesn’t take away any related pairs from eval, so the same proof as before still works.

(c) In general, is there any way we could cause eval_progress to fail if we took away one or more constructors from the original eval relation given in question 15? Write yes or no and briefly (1 sentence) explain your answer. Answer: Yes. For example, if we take away all the constructors, then any non-value term gives us a counter-example.

Grading scheme: 3 points for each part. 1 point for correct yes/no answer. 1 additional point for a partial or confusing (but “somewhat correct”) explanation.