CIS 500 — Software Foundations
Midterm I

October 10, 2007

Name: ____________________________________________

Email: ____________________________________________

<table>
<thead>
<tr>
<th></th>
<th>Score</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td></td>
</tr>
<tr>
<td>13</td>
<td></td>
</tr>
<tr>
<td>14</td>
<td></td>
</tr>
<tr>
<td>15</td>
<td></td>
</tr>
<tr>
<td>16</td>
<td></td>
</tr>
<tr>
<td>17</td>
<td></td>
</tr>
<tr>
<td>Total</td>
<td></td>
</tr>
</tbody>
</table>
Instructions

• This is a closed-book exam: you may not use any books or notes.

• You have 80 minutes to answer all of the questions.

• Questions vary significantly in difficulty, and the point value of a given question is not always exactly proportional to its difficulty. Do not spend too much time on any one question.

• Partial credit will be given. All correct answers are short. The back side of each page may be used as a scratch pad.

• Good luck!
1. (2 points) Consider the following \texttt{Fixpoint} definition in Coq.

\begin{verbatim}
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.
\end{verbatim}

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

2. (2 points) What does

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

print?
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

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

prints:

\[ = ([\text{one, two}], [\text{no, no}]) \]

\[ : \text{list nat * list yesno} \]

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).

\[
\text{Fixpoint unzip (X Y : Set) (l : list (X*Y)) } \{\text{struct l}\} \]: _____________________________ :=

match l with
| nil =>
  ________________________________
| cons (x,y) t =>
  match unzip X Y t with
    ________________________________
end
end.

3
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.
5. (4 points) Briefly explain what the `assert` tactic does.

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

   ```coq
   Inductive evenI : nat -> Prop :=
   | even_zero : evenI 0
   | even_SS : forall n:nat, evenI n -> evenI (S (S n)).
   ```

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

   ```coq
   Lemma l : forall n, evenI n.
   Proof.
   intros n. induction n.
   Case "0". simpl. apply even_zero.
   Case "S".
   ...
7. (6 points) Recall the definition of `plus`:

```coq
Fixpoint plus (m : nat) (n : nat) {struct m} : nat :=
  match m with
  | O => n
  | S m' => S (plus m' n)
  end.
```

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

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

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

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

(c) Briefly (1 or two sentences) explain the difference.
Inductively Defined Sets

8. (8 points) Consider the following inductive definition:

```
Inductive foo (X:Set) : Set :=
  | c1 : list X -> foo X -> foo X
  | c2 : foo X.
```

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

```
foo_ind :
  forall (X : Set) (P : foo X -> Prop),
  (forall (l : list X) (f : foo X),
    ___________________ -> ___________________ )
  -> ________________________
  -> forall f : foo X, ________________________
```

9. (6 points) 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, ________________________
```

What is the definition of `myset`?
Inductively Defined Propositions

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

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

Now consider the following inductively defined proposition:

    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)
    | c3 : forall t n, p t n -> p t (S n).

Describe, in English, the conditions under which the proposition p t n is provable.
11. (4 points) Suppose we give Coq the following definition:

\[
\text{Inductive } R : \text{nat} \to \text{nat} \to \text{nat} \to \text{Prop} :=
\begin{align*}
| \text{c1} & : R \text{ zero zero zero} \\
| \text{c2} & : \text{forall } m \text{ n o, } R \text{ m n o } \to R \text{ S m n S o)} \\
| \text{c3} & : \text{forall } m \text{ n o, } R \text{ m n o } \to R \text{ m S n S o)} \\
| \text{c4} & : \text{forall } m \text{ n o, } R \text{ S m S n S S o) } \to R \text{ m n o} \\
| \text{c5} & : \text{forall } m \text{ n o, } R \text{ m n o } \to R \text{ n m o}.
\end{align*}
\]

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

(a) R one one two

(b) R two two six

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.

13. (2 points) If we dropped constructor c4 from the definition of R, would the set of provable propositions change? Write yes or no.
14. (5 points) Complete the following definition of existential quantification as an inductive proposition in Coq.

\[
\text{Inductive ex (X : Type) (P : X \to Prop) : Prop :=}
\]
15. (6 points) In the lectures, we have been working with a very simple programming language whose terms consist of just \texttt{plus} and constants. Here is an equally simple language whose terms are just the boolean constants \texttt{true} and \texttt{false} and a conditional expression:

\begin{verbatim}
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).
\end{verbatim}

Which of the following propositions are provable? (Write \texttt{yes} or \texttt{no} next to each.)

(a) \hspace{1cm} \texttt{eval tm_false tm_false} \hspace{1cm} \texttt{yes}

(b) \hspace{1cm} \texttt{eval (tm_if tm_true (tm_if tm_true tm_true tm_true) tm_true)} \hspace{1cm} \texttt{no}

(c) \hspace{1cm} \texttt{eval (tm_if (tm_if tm_true tm_true tm_true) tm_false) (tm_if tm_true tm_false)} \hspace{1cm} \texttt{yes}
16. (4 points) Suppose we want to add a “short circuit” to the evaluation relation so that it can recognize when the \textit{then} and \textit{else} branches of a conditional are the same value (either \texttt{tm\_true} or \texttt{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:

\[
\text{eval}
\begin{align*}
  \text{tm\_if} \\
  \hspace{1em} \text{tm\_if tm\_true tm\_true tm\_true} \\
  \hspace{2em} \text{tm\_false} \\
  \hspace{3em} \text{tm\_false}
\end{align*}
\text{tm\_false}
\]

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

\[
| \text{E\_ShortCircuit} : \text{forall } \texttt{__________________________________________}, \\
  \text{________________________________________________________} |
\]
17. (9 points) It can be shown that the determinism and progress theorems for the `eval` relation in the lecture notes...

    Theorem eval_deterministic :
        partial_function _ eval.

    Theorem eval_progress : forall t,
        value t / (exists t', 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.

(b) Does `eval_progress` still hold? Write `yes` or `no` and briefly (1 sentence) explain your answer.

(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.