CIS 500: Software Foundations

Midterm I

October 3, 2019

Name (printed): ________________________________

Username (PennKey login id): ________________________________

My signature below certifies that I have complied with the University of Pennsylvania's Code of
Academic Integrity in completing this examination.

Signature: ________________________________ Date: ________________________________

**Directions:** This exam contains both standard and advanced-track questions. Questions
with no annotation are for *both* tracks. Other questions are marked “Standard Track Only” or
“Advanced Track Only.” *Do not waste time (or confuse the graders) by answering questions
intended for the other track.*

Mark the box of the track you are following.

☐ Standard ☐ Advanced

Before beginning the exam, please write your PennKey (login ID) at the top of each even-numbered
page (so that we can find things if a staple fails!).
(11 points) Put an X in the True or False box for each statement.

(a) There exists a proposition \( P \) such that the proposition \( \neg \neg P \iff P \) is provable.

\[
\begin{array}{ccc}
\Box & \text{True} & \Box & \text{False}
\end{array}
\]

(b) For every number \( n \), the boolean computation \texttt{evenb n} is reflected in the truth of the proposition \( \exists k, n = \text{double} k \).

\[
\begin{array}{ccc}
\Box & \text{True} & \Box & \text{False}
\end{array}
\]

(c) Suppose we have assumption \( H : P \rightarrow Q \rightarrow R \rightarrow S \) and the current goal is \( S \). If we do \texttt{apply H}, then the goal will change to \( P \rightarrow Q \rightarrow R \).

\[
\begin{array}{ccc}
\Box & \text{True} & \Box & \text{False}
\end{array}
\]

(d) The type
\[
\text{Inductive Foo : Type :=}
| \text{MakeFoo (f : Foo)}.
\]

has an infinite number of elements.

\[
\begin{array}{ccc}
\Box & \text{True} & \Box & \text{False}
\end{array}
\]

(e) The result of \texttt{Compute (In 1 [1;2])} is \texttt{True}.

\[
\begin{array}{ccc}
\Box & \text{True} & \Box & \text{False}
\end{array}
\]

(f) There are no empty types in Coq. In other words, for any type \( A \), there is some Coq expression that has type \( A \).

\[
\begin{array}{ccc}
\Box & \text{True} & \Box & \text{False}
\end{array}
\]

(g) Coq’s termination checker uses a conservative syntactic condition to guarantee that all \texttt{Fixpoint} definitions define total functions: there must be one of the function’s arguments that is “decreasing” at each recursive call in the function’s body, in the sense that the argument to the recursive call is a “strict subterm” of the corresponding argument to the function itself.

\[
\begin{array}{ccc}
\Box & \text{True} & \Box & \text{False}
\end{array}
\]

(h) \( f = g \rightarrow \forall x, f x = g x \) requires an additional axiom to prove in Coq.

\[
\begin{array}{ccc}
\Box & \text{True} & \Box & \text{False}
\end{array}
\]
(i) \((\forall x, f x = g x) \rightarrow f = g\) requires an additional axiom to prove in Coq.

\[ \square \text{True} \quad \square \text{False} \]

(j) Using the tactic \texttt{injection} H when \(H : \text{double } n = \text{double } m\) is in the proof context will introduce \(n = m\) as a hypothesis.

\[ \square \text{True} \quad \square \text{False} \]

(k) We’ve seen situations where using the tactic \texttt{rewrite} H will have exactly the same effect as using the tactic \texttt{apply} H.

\[ \square \text{True} \quad \square \text{False} \]
(a) fun (x y z : nat) => 1 + z

(b) fun (b : bool) => if true then 1 else b

(c) forall (X : Type) (x y : X), x * y = 5

(d) forall (X : Type) (x y : X), x = y

(e) fold plus

(f) forall (X Y : Prop), X -> Y

(g) forall (x : nat) (y : Prop), x -> y

(h) fun x => exists y, y <= x
(11 points) For each of the following propositions, check “not provable” if it is not provable (in Coq’s core logic, without additional axioms), “needs induction” if it is provable only using induction, or “easy” if it is provable without using induction and without additional lemmas.

(a) $\text{In } 3 \ [1;2;3;4;5]$  
   □ not provable   □ needs induction   □ easy

(b) $\forall s, \text{In } 3 \ ([1;2;3] \ ++ \ s)$  
   □ not provable   □ needs induction   □ easy

(c) $\forall s, \text{In } 3 \ (s \ ++ \ [1;2;3])$  
   □ not provable   □ needs induction   □ easy

(d) $\exists s, \text{In } 3 \ (s \ ++ \ [1;2;3])$  
   □ not provable   □ needs induction   □ easy

(e) $\exists (x \ y : \text{list} \ \text{nat}), \ x \ ++ \ y = y \ ++ \ x$  
   □ not provable   □ needs induction   □ easy

(f) $\forall n, n+5 \leq n+6$  
   □ not provable   □ needs induction   □ easy

(g) $\forall f \ g, (\forall x, f \ x = g \ x) \implies f = g$  
   □ not provable   □ needs induction   □ easy

(h) $\forall x \ y, x \ * \ y = y \ * \ x$  
   □ not provable   □ needs induction   □ easy

(i) $\forall P : \text{Prop}, P \ \text{\slash} \ \sim P$  
   □ not provable   □ needs induction   □ easy

(j) $\forall P : \text{Prop}, P \implies \sim \sim P$  
   □ not provable   □ needs induction   □ easy

(k) $\forall P : \text{Prop}, P$  
   □ not provable   □ needs induction   □ easy
(a) \text{option Prop}

(b) \text{forall (X : Type), (X -> Prop) -> list X -> Prop}

(c) \text{nat -> nat -> nat}

(d) \text{list (nat -> nat -> nat)}

(e) \text{forall (A : Type), A}

(f) \text{1 <= 3}

(g) \text{forall (A B : Type), A -> B -> A}

(h) \text{bool * Prop}
5 (6 points) For each of the following, select the tactics which will prove the theorem.

\[\text{Inductive ev : nat -> Prop :=}
  \mid \text{ev}_0 : \text{ev } 0
  \mid \text{ev}_{SS} (n : \text{nat}) (H : \text{ev } n) : \text{ev } (S \ (S\ n)).\]

\[\text{Inductive le : nat -> nat -> Prop :=}
  \mid \text{le}_n : \forall n, \text{le } n\ n
  \mid \text{le}_{S} : \forall n \ m, (\text{le } n\ m) \rightarrow (\text{le } n\ (S\ m)).\]

\text{Notation "m \leq n" := (le m n).}

(a) Which of these tactic scripts would prove the following theorem (there may be zero, one, or
more; check as many as apply)?

\[\text{Theorem even\_term :}
  \forall n, \text{ev } n \rightarrow \text{ev } (S \ (S \ (S \ (S\ n))))\].

- \[\square\ \text{intros n H. apply (ev}_{SS} \ (ev}_{SS} \ H)).\]
- \[\square\ \text{intros n H. apply (ev}_{SS} \ (S \ (S\ n)) \ (ev}_{SS} \ n\ H)).\]
- \[\square\ \text{intros n H. apply (ev}_{SS} \ n \ (ev}_{SS} \ (S \ (S\ n)) \ H)).\]
- \[\square\ \text{intros n H. apply (ev}_{SS} \ (S \ (S \ (S\ n)))
    \ (ev}_{SS} \ (S \ (S\ n)) \ (ev}_{SS} \ n \ (ev}_{SS} \ n\ H))).\]
- \[\square\ \text{intros n H. apply (ev}_{SS} \ (ev}_{SS} \ (ev}_{SS} \ H))).\]

(b) Which of these tactic scripts would prove the following theorem (check as many as apply)?

\[\text{Theorem le\_term :}
  1 \leq 3\].

- \[\square\ \text{apply (le}_{S} \ 1 \ 1 \ (le}_{S} \ 2 \ 1 \ (le}_{n} \ 1))).\]
- \[\square\ \text{apply (le}_{S} \ (le}_{S} \ le}_{n))).\]
- \[\square\ \text{apply (le}_{S} \ 2 \ 1 \ (le}_{S} \ 1 \ 1 \ (le}_{n} \ 1))).\]
- \[\square\ \text{apply (le}_{n} \ (le}_{S} \ le}_{S))).\]
- \[\square\ \text{apply (le}_{S} \ 1 \ 2 \ (le}_{S} \ 1 \ 1 \ (le}_{n} \ 1))).\]
This problem asks you to translate mathematical ideas from English into Coq.

(a) (4 points) Complete the following definition of a property characterizing the prime natural numbers (i.e., those numbers that cannot be expressed as the product of two other numbers, both strictly greater than 1).

\[
\text{Definition prime (n: nat) : Prop :=}
\]
(b) (4 points) A binary tree with labels of type A is either empty or a branch that has some value of type A along with two binary trees of type A as children. Its formal definition is provided on page 2 of the appendix. In the following example, exTree on the left corresponds to the tree on the right:

```
Example exTree : tree nat :=
  Branch 5
    (Branch 2
      (Branch 1 Empty Empty)
      (Branch 4 Empty Empty))
    (Branch 9
      Empty
      (Branch 7 Empty Empty)).
```

Now, suppose we define a predicate EveryLabel of the form:

```
Fixpoint EveryLabel {A : Type} (P : A -> Prop) (t: tree A) : Prop :=
  match t with
  | Empty => True
  | Branch x l r => P x /
                   EveryLabel P l /
                   EveryLabel P r
  end.
```

For example,

```
EveryLabel (fun x => x<=10) exTree
```

is provable, while

```
not (EveryLabel (fun x => ev x) exTree)
```

is not.

A tree is called a binary search tree if for all nodes (Branch x l r) in the tree, all values stored in the left subtree l are less than or equal to the node’s value x, and all values stored in the right subtree r are greater than x. For example, the tree shown above is not a binary search tree, but it will become one if we change the label 7 to 10.

Use EveryLabel to complete the definition of an inductive property BST that characterizes binary search trees.

```
Inductive BST : tree nat -> Prop :=
  | Empty_BST : BST Empty
  | Branch_BST :
```

```
(c) (5 points) Continuing with binary trees, in this part, we will next define a relation \texttt{mirror} that relates two trees that are “mirror images” of each other. For example, the following should be provable:

\begin{verbatim}
Example mirror_eg :
mirror (Branch 1 Empty (Branch 2 (Branch 3 Empty Empty) Empty))
  (Branch 1 (Branch 2 Empty (Branch 3 Empty Empty)) Empty).
\end{verbatim}

This says that the following two mirrored trees are mirrors of each other:

\begin{center}
\begin{tikzpicture}
  \node (1) at (0,0) {1}
  \node (2) at (-1,-1) {2}
  \node (3) at (-2,-2) {3}
  \draw (1) -- (2);
  \draw (1) -- (3);
\end{tikzpicture}
\quad \quad \begin{tikzpicture}
  \node (1) at (0,0) {1}
  \node (2) at (1,-1) {2}
  \node (3) at (2,-2) {3}
  \draw (1) -- (2);
  \draw (1) -- (3);
\end{tikzpicture}
\end{center}

Here is a skeleton for \texttt{mirror}; please fill in the missing part.

\begin{verbatim}
Fixpoint mirror {A: Type} (t1 t2 : tree A) : Prop :=
  match t1,t2 with
  (* FILL IN HERE *)
end.
\end{verbatim}
(d) (7 points) Continuing with binary trees, here is a way of representing a “path” in a binary tree.

    Inductive path :=
    | Stop
    | GoLeft (p : path)
    | GoRight (p : path).

In this problem, we will define a ternary relation \texttt{valueAtPath} that relates a tree and a path to the value (if any) picked out by following that path in the tree. For example, the following should both be provable, where \texttt{t} is the tree pictured on the right:

    Definition \texttt{t} :=
        Branch 1 Empty (Branch 2 (Branch 3 Empty Empty) Empty).

    Example \texttt{VAP\_eg1} : valueAtPath \texttt{t} Stop 1.

    Example \texttt{VAP\_eg2} : valueAtPath \texttt{t} (GoRight (GoLeft Stop)) 3.

Here is a skeleton for \texttt{valueAtPath}; please fill in the missing bits.

    Inductive \texttt{valueAtPath} {A: Type} : tree A -> path -> A -> Prop :=
    | Here : forall (t1 t2: tree A) (a: A),
        (* FILL IN HERE *)

    | InLeft : forall (t1 t2: tree A) (b: A) (p: path) (a: A),
        (* FILL IN HERE *)

    | InRight : forall (t1 t2: tree A) (b: A) (p: path) (a: A),
        (* FILL IN HERE *)
Recall the polymorphic \texttt{fold} function.

\begin{verbatim}
Fixpoint fold \{X Y\} (f : X \to Y \to Y) (l : list X) (b : Y) : Y :=
  match l with
  | [] => b
  | h :: t => f h (fold f t b)
end.
\end{verbatim}

The arguments to \texttt{fold} are an aggregating function \(f : X \to Y \to Y\), a list \(l : \text{list} X\), and an initial value \(b : Y\); given these, it “folds through the list” applying the function \(f\) on successive elements. Here is an example use of \texttt{fold}, showing several evaluation steps.

\begin{verbatim}
fold app [[]1, [], [2,3], [4]] []
= app [1] (fold app [[]1, [2,3], [4]] [])
= ... 
= [1,2,3,4]
\end{verbatim}

Suppose we define a variant \texttt{reverse_fold} that folds a given list while reversing its elements.

\begin{verbatim}
Fixpoint reverse_fold \{X Y\} (f : X \to Y \to Y) (l : list X) (b : Y) : Y :=
  match l with
  | [] => b
  | h :: t => reverse_fold f t (f h b)
end.
\end{verbatim}

For the same \(f\) and \(l\) as before, the output of \texttt{reverse_fold} would be:

\begin{verbatim}
reverse_fold app [[]1, [], [2,3], [4]] []
= reverse_fold app [[]1, [2,3], [4]] (app [] [])
= reverse_fold app [[]2,3, [4]] (app [] (app [1] []))
= reverse_fold app [[4]] (app [2,3] (app [] (app [1] [])))
= ... 
= [4,2,3,1]
\end{verbatim}

Finally, recall the polymorphic \texttt{rev} function that reverses the elements of a list.

\begin{verbatim}
Fixpoint rev \{X\} (l : list X) : list X :=
  match l with
  | [] => []
  | h :: t => rev t ++ [h]
end.
\end{verbatim}

(The definitions of \texttt{fold}, \texttt{reverse_fold}, and \texttt{rev} can also be found on page 2 in the appendix.)

The following theorem intuitively says that “the result of applying \texttt{reverse_fold} on a list \(l\) is the same as the result of applying \texttt{fold} to the reverse of list \(l\).”

**Theorem** \texttt{reverse_fold_rev_fold} :
\[\forall X Y \forall (f : X \rightarrow Y \rightarrow Y) (b : Y) (l : \text{list} X), \text{fold} f l b = \text{fold} f (\text{rev} l) b.\]

On the next page, write a careful, informal proof of this theorem. If your proof uses induction, be sure to spell out the induction hypothesis explicitly. You may find it helpful to define an auxiliary lemma characterizing how \texttt{fold} works when applied to a list of the form \texttt{11 ++ 12}. If so, please give just the statement of the lemma—there is no need to prove it (make sure it is true, though!). You can write this auxiliary lemma anywhere in the blank space provided (clearly marked, please).
Theorem reverse_fold_rev_fold :
  forall X Y (f : X -> Y -> Y) (b : Y) (l : list X),
  reverse_fold f l b = fold f (rev l) b.
Use this space for scratch work that you don't want graded. If you write something here that you do want graded, make sure there is a very clear pointer from the earlier page where you ran out of space.
For Reference

Numbers

Inductive nat : Type :=
| O : nat
| S : nat -> nat.

Fixpoint plus (n : nat) (m : nat) : nat :=
match n with
| O => m
| S n' => S (plus n' m)
end.
Notation "x + y" := (plus x y)(at level 50, left associativity) : nat_scope.

Fixpoint mult (n : nat) (m : nat) : nat :=
match n with
| 0 => 0
| S n' => m + (mult n' m)
end.
Notation "x * y" := (mult x y)(at level 40, left associativity) : nat_scope.

Fixpoint double (n:nat) :=
match n with
| O => O
| S n' => S (S (double n'))
end.

Fixpoint evenb (n : nat) : bool :=
match n with
| O => true
| S O => false
| S (S n') => evenb n'
end.

Inductive le : nat -> nat -> Prop :=
| le_n : forall n, le n n
| le_S : forall n m, (le n m) -> (le n (S m)).
Notation "m <= n" := (le m n).

Inductive ev : nat -> Prop :=
| ev_0 : ev 0
| ev_SS (n : nat) (H : ev n) : ev (S (S n)).

Pairs

Inductive prod (A : Type) (B : Type) :=
| pair : A -> B -> prod A B.
Arguments pair {A B}.
Notation "A * B" := (prod A B) (at level 40, left associativity) : type_scope.
Notation "( x , y )" := (pair x y) (at level 0) : core_scope.
Lists

Inductive list (X:Type) : Type :=
| nil : list X
| cons : X -> list X -> list X.

Arguments nil {X}.
Arguments cons {X} _ _.
Notation "[ ]" := nil.
Notation "x :: l" := (cons x l) (at level 60, right associativity).
Notation "[ x ; .. ; y ]" := (cons x .. (cons y nil) ..).

Fixpoint app {X : Type} (l1 l2 : list X) : (list X) :=
match l1 with
| [] => l2
| h :: t => h :: (app t l2)
end.
Notation "x ++ y" := (app x y) (at level 60, right associativity).

Fixpoint rev {X} (l : list X) : list X :=
match l with
| [] => []
| h :: t => rev t ++ [h]
end.

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

Fixpoint reverse_fold {X Y} (f : X -> Y -> Y) (l : list X) (b : Y) : Y :=
match l with
| [] => b
| h :: t => reverse_fold f t (f h b)
end.

Fixpoint In {A : Type} (x : A) (l : list A) : Prop :=
m_match l with
| [] => False
| x' :: l' => x' = x / In x l'
end.

Binary Trees

Inductive tree (A : Type) :=
| Empty : tree A
| Branch : A -> tree A -> tree A -> tree A.

Arguments Empty {A}.
Arguments Branch {A} _ _ _.