CIS 500: Software Foundations Midterm I
October 2, 2018

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 booklet contains both the standard and advanced track questions. Questions with no annotation are for both tracks. Other questions are marked “Standard Only” or “Advanced Only.” Do not waste time answering questions intended for the other track.

Put an X in the box of the track you are following. (If you are following one track but want to move to the other now, please mark the box for the track you want to be on and write a note on this page telling us that we should switch you to this track.)

□ Standard □ Advanced
1. (6 points) Put an X in the True or False box for each statement.

(a) All functions defined in Coq via Fixpoint must terminate on all inputs.

□ True    □ False

(b) If foo is an inductive type and f : nat -> foo is one of its constructors, and if H : f x = f y is an assumption in the current context, then injection H will introduce an assumption that x = y.

□ True    □ False

(c) If X is an inductive type with at least one constructor, then there is at least one constructor expression e such that e : X.

□ True    □ False

(d) If S (n + m) = n + S m is the goal in the current proof state, then reflexivity will solve the goal.

□ True    □ False

(e) For any number n, if the goal in the current proof state is S n = 0, then using the tactic discriminate will solve the goal.

□ True    □ False

(f) For any propositions P and Q, if H : P -> Q is an assumption in the current context and the goal in the current proof state is P, then apply H will solve the goal.

□ True    □ False
2. (16 points) Write the type of each of the following Coq expressions (write “ill typed” if an expression does not have a type). For part e you can find the definition of \texttt{repeat} on page \ref{2} in the appendix.

(a) $1 + 1 = 2$

(b) \texttt{fun} ($x$ $y$ : nat) => $x + 1 = y$

(c) \texttt{forall} ($X$ : Type) ($x$ : $X$), $x$

(d) \texttt{fun} ($m$ : nat) => \texttt{fun} ($n$ : nat) => $m * m + n$

(e) \texttt{repeat} 3

(f) \texttt{forall} ($X$ : Type), \texttt{forall} ($Y$ : Type), \texttt{forall} ($x$ : $X$), $x = x$

(g) \texttt{fun} ($m$ : nat) => $m :: m :: m$

(h) \texttt{match} [1;2] with [] => \texttt{true} | $x :: l$ => $x + 1$ end
3. [Standard Track Only] (16 points) For each of the types below, write a Coq expression that has that type or write “empty” if there are no such expressions.

(a) list (list nat)

(b) nat -> list nat -> nat

(c) bool -> bool -> bool

(d) forall (X : Type) (l1 l2 : list X), list X

(e) forall (X : Type), X -> X -> X

(f) Prop -> (bool -> bool)

(g) Prop * Prop

(h) list (forall X, X -> list X)
4. [Advanced Track Only] (16 points)

Recall the polymorphic `fold` and `++` functions (see page 2 in the appendix).

The following theorem intuitively says that “`fold` commutes with `++`”—that is, if we have two lists, `l1` and `l2`, a function `f`, and a starting value `b`, then either we can append the lists together and then fold `f` over them or else we can fold `f` over `l2` and then fold `f` again over `l1`, using the previous result as the starting point.

```coq
Theorem app_comm_fold :
  forall {X Y} (f: X->Y->Y) l1 l2 b,
  fold f (l1 ++ l2) b = fold f l1 (fold f l2 b).
```

Write a careful, informal proof of this theorem. If your proof uses induction, be sure to spell out the induction hypothesis explicitly.
5. This problem asks you to translate mathematical ideas from English into Coq.

(a) (5 points) The concept of “prefix of a list” can be defined as an inductive relation as follows:

\[
\text{Inductive \textit{prefix} \{X : \text{Type}\} : \text{list} X \rightarrow \text{list} X \rightarrow \text{Prop} :=}
\]

| \text{\textit{prefix} \textit{nil} : \forall l, \text{\textit{prefix}} [\] l} |
| \text{\textit{prefix} \textit{cons} : \forall t l h, \text{\textit{prefix}} t l \rightarrow \text{\textit{prefix}} (h::t) (h::l).} |

For example, these are all the prefixes of \([1;2;3]\) (i.e., all the lists \(ls\) such that \(\text{\textit{prefix}} \, ls\ \ [1;2;3]\) is provable):

\[
[\]
[1]
[1;2]
[1;2;3]
\]

Conversely, a \textit{suffix} of a list is a substring that occurs at the end of the larger string. For example, here are all the suffixes of \([1;2;3]\):

\[
[\]
[3]
[2;3]
[1;2;3]
\]

Complete the following inductive definition of the suffix relation, where \textit{suffix} \(s\ \ l\) indicates that \(s\) is a suffix of \(l\).

\[
\text{Inductive \textit{suffix} \{A : \text{Type}\} : \text{list} A \rightarrow \text{list} A \rightarrow \text{Prop} :=}
\]
(b) (8 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 4 of the appendix. For example, the definition

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

represents the following binary tree:

```
      5
     /|
    2 9
   /|
  1 4 7
```

A subtree of a binary tree is either the tree itself or a subtree of its child. For example, here are all the subtrees of $\text{exTree}$:

```
Empty 1 4 7 1 4 7 1 4 7
```

(continued...)
Give an inductive definition of the subtree relation on binary trees, where \( \text{subtree } s \ t \) indicates that \( s \) is a subtree of \( t \).

\[
\text{Inductive subtree } \{A : \text{Type}\} : \text{tree } A \to \text{tree } A \to \text{Prop} :=
\]
6. (4 points) Recall the definition of the \( \text{In} \) relation between values of type \( A \) and lists with elements of type \( A \):

\[
\text{Fixpoint In \{A : Type\} (x : A) (l : list A) : Prop :=}
\begin{align*}
 & \text{match l with} \\
 & \quad | \emptyset \Rightarrow \text{False} \\
 & \quad | x' :: l' \Rightarrow x = x' \lor \text{In} x l'
\end{align*}
\]

Suppose we want to define an analogous relation \( \text{NotIn} \) such that \( \text{NotIn} x l \) is true just when \( x \) does not appear in \( l \).

Below are several candidates for how we might define this relation. Check the boxes next to those that are correct—i.e., the ones where \( \text{NotIn} x l \) is provable exactly when \( \lnot (\text{In} x l) \) is.

- \( \Box \) \begin{align*}
\text{Fixpoint NotIn \{A : Type\} (x : A) (l : list A) : Prop :=}
\quad & \text{match l with} \\
\quad & \quad | \emptyset \Rightarrow \text{False} \\
\quad & \quad | x' :: l' \Rightarrow x' \neq x \land \text{NotIn} x l'
\end{align*}

- \( \Box \) \begin{align*}
\text{Inductive NotIn \{A : Type\} : A \rightarrow list A \rightarrow Prop :=}
\quad & \text{NotInHere : forall x x' l', x' \neq x' \rightarrow NotIn x (x' :: l')} \\
\quad & \text{NotInThere : forall x x' l', NotIn x l' \rightarrow NotIn x (x' :: l')}.
\end{align*}

- \( \Box \) \begin{align*}
\text{Definition NotIn \{A : Type\} := fun (x : A) (l : list A) =>}
\quad & \forall x', \text{In} x' l \rightarrow x \neq x'.
\end{align*}

- \( \Box \) \begin{align*}
\text{Definition NotIn \{A : Type\} := fun (x : A) (l : list A) =>}
\quad & \forall x', \text{In} x' l \setminus x \rightarrow x'.
\end{align*}
7. (9 points)

(a) Suppose Coq’s current goal state looks like this:

```
P, Q, R : Prop
H1 : P \lor Q
H2 : Q \land R
H3 : \forall S : Prop, S \rightarrow P \lor S
=================================
P \lor \neg Q
```

i. If we give the command “`split`”, what will happen?

- □ Error
- □ Nothing (no error, but no change to the state)
- □ “No more subgoals”
- □ Goal changes to P
- □ Goal changes to \neg Q

ii. If we give the command “`apply H3`”, what will happen?

- □ Error
- □ Nothing (no error, but no change to the state)
- □ “No more subgoals”
- □ Goal changes to P\lor S
- □ Goal changes to P
- □ Goal changes to \neg Q
- □ Goal changes to Q

iii. If we give the command “`destruct H1`”, what will happen?

- □ Error
- □ Nothing (no error, but no change to the state)
- □ “No more subgoals”
- □ H1 is replaced by two hypotheses, H : P and H1 : Q
- □ Current goal is replaced by two subgoals, one with hypothesis H : P and one with hypothesis H : Q
(b) Suppose Coq’s current goal state looks like this:

\[
\begin{align*}
m, n : \text{nat} \\
H : \forall x, x + m = m + x \\
R : \forall a \ b \ c, S a = b + S c \rightarrow a = b + c \\
\text{============================} \\
S (m + n) = n + S m
\end{align*}
\]

i. If we give the command “\texttt{induction n as [| n' IHn']\textquoteright}”, we will get two subgoals. What will the second one look like?

- New hypothesis \texttt{IHn'} : \(S (m + n') = n' + S m\)
  and the goal changes to \(S (m + S n') = S n' + S m\)
- New hypothesis \texttt{IHn'} : \(\forall m, S (m + n') = n' + S m\)
  and the goal changes to \(\forall m, S (m + S n') = S n' + S m\)
- New hypothesis \texttt{IHn'} : \(S (m + S n') = S n' + S m\)
  and the goal changes to \(\forall m, S (m + S n') = S n' + S m\)
- New hypothesis \texttt{IHn'} : \(\forall m, S (m + S n') = S n' + S m\)
  and the goal changes to \(\forall m, S (m + n') = n' + S m\)

ii. If we give the command “\texttt{rewrite -> H\textquoteright}”, what will happen?

- Error
- Nothing (no error, but no change to the state)
- Goal changes to \(S (n + m) = n + S m\)
- Goal changes to \(S (n + m) = S m + n\)
- Goal changes to \(S (m + n) = S m + n\)

iii. If we give the command “\texttt{apply R\textquoteright}”, what will happen?

- Error
- Nothing (no error, but no change to the state)
- Goal changes to \(m + n = n + m\)
- Goal changes to \(S (S (m + n)) = n + S (S m)\)
- No more subgoals
8. (16 points) In this problem you will help write a simple regular expression matching function, `re_match`. The expression `re_match s re` should yield `true` if and only if the “string” `s` (a list of `nats`) matches the regular expression `re`. For example:

```
re_match [1; 3] (App (Union (Char 1) (Char 2)) (Char 3)) = true
re_match [1; 2; 3] (App (Union (Char 1) (Char 2)) (Char 3)) = false
```

Complete the recursive definition of the function `re_match` on the following page, filling in the branches for `EmptySet`, `Char`, `App` and `Union`. Your answers may use any function defined in the appendix. In particular, the following functions might be useful:

- `list_nat_eq : list nat -> list nat -> bool` checks whether two lists are equal.
- `existsb : forall X : Type, (X -> bool) -> list X -> bool` checks whether at least one element in a list `l : list X` satisfies a boolean property `p : X -> bool`.
- `forallb : forall X : Type, (X -> bool) -> list X -> bool` checks whether all elements of a list `l : list X` satisfy a boolean property `p : X -> bool`.
- `splits : forall X : Type, list X -> list (list X * list X)` enumerates all possible ways to split a list `s` into two lists `(s1, s2)` such that `s1 ++ s2 = s`. For example:

```
splits [1; 2; 3] = [ ([], [1; 2; 3]);
    ([1], [2; 3]);
    ([1; 2], [3]);
    ([1; 2; 3], []) ]
```

The function `splits_many` that is used in the `Star` case is a kind of `n`-way generalization of `splits`. You don’t need to understand it in detail, but it enumerates all possible ways to split a list `s` into one or more nonempty lists `s1,...,sn` such that `s1 ++ ... ++ sn = s`. For example:

```
splits_many [1; 2; 3] = [ [[1]]; [2]; [3]];  
    [[1]; [2; 3]];  
    [[1; 2]; [3]];  
    [[1; 2; 3]]    ]
```
Fixpoint re_match (s : list nat) (re : reg_exp) : bool :=
  match re with
  | EmptySet => (* FILL IN HERE: * )

  | EmptyStr => list_nat_eq s []

  | Char n => (* FILL IN HERE: * )

  | App re1 re2 => (* FILL IN HERE: * )

  | Union re1 re2 => (* FILL IN HERE: * )

  | Star re' =>
    list_nat_eq s [] ||
    existsb (fun ss' => forallb (fun s' => re_match s' re') ss')
    (splits_many s)

  end.
For Reference

Booleans

Inductive bool : Type :=
| true
| false.

Definition negb (b : bool) : bool :=
  match b with
  | true => false
  | false => true
  end.

Definition andb (b c : bool) : bool :=
  match b with
  | true => c
  | false => false
  end.

Definition orb (b c : bool) : bool :=
  match b with
  | false => c
  | true => true
  end.

Notation "b && c" := (andb b c) (at level 50, left associativity) : bool_scope.
Notation "b || c" := (orb b c) (at level 40, left associativity) : bool_scope.

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
  | O => 0
  | S n' => m + (mult n' m)
  end.
Notation "x * y" := (mult x y)(at level 40, left associativity) : nat_scope.
Fixpoint eqb (n m : nat) : bool :=
  match n, m with
  | 0, 0 => true
  | S n', S m' => eqb n' m'
  | _, _ => false
end.
Notation "x =? y" := (eqb x y) (at level 70) : nat_scope.

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 : forall n : nat, 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 repeat (n count : nat) : list nat :=
  match count with
  | 0 => nil
  | S count' => n :: (repeat n count')
end.
Fixpoint In {A : Type} (x : A) (l : list A) : Prop :=
  match l with
  | [] => False
  | x' :: l' => x' = x \ In x l'
  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 forallb {X : Type} (test : X -> bool) (l : list X) : bool :=
  match l with
  | [] => true
  | x :: l' => andb (test x) (forallb test l')
  end.

Fixpoint existsb {X : Type} (test : X -> bool) (l : list X) : bool :=
  match l with
  | [] => false
  | x :: l' => orb (test x) (existsb test l')
  end.

Fixpoint list_nat_eq (s1 s2 : list nat) : bool :=
  match s1, s2 with
  | n1 :: t1, n2 :: t2 => (n1 =? n2) && list_nat_eq t1 t2
  | [], [] => true
  | _, _ => false
  end.

(* Enumerate all ways to split a list into two. *)
Fixpoint splits {X : Type} (s : list X) : list (list X * list X) :=
  match s with
  | [] => [([], [])]
  | h :: t => ([], s) :: map (fun '(s1, s2) => (h :: s1, s2)) (splits t)
  end.

(* Enumerate all ways to split a list into _any number_ of _nonempty_ lists. *)
Fixpoint splits_many {X : Type} (s : list X) : list (list (list X)) :=
  match s with
  | [] => []
  | h :: [] => [[s]]
  | h :: t =>
    map (fun t' => [h] :: t') (splits_many t) ++
    map (fun t' =>
      match t' with
      | [] => t'
      | u :: w => (h :: u) :: w
      end) (splits_many t)
  end.
Regular Expressions

Inductive reg_exp {T : Type} : Type :=
| EmptySet : reg_exp
| EmptyStr : reg_exp
| Char : T -> reg_exp
| App : reg_exp -> reg_exp -> reg_exp
| Union : reg_exp -> reg_exp -> reg_exp
| Star : reg_exp -> reg_exp.

Inductive exp_match {T : Type} : list T -> reg_exp -> Prop :=
| MEmpty : exp_match [] EmptyStr
| MChar : forall x, exp_match [x] (Char x)
| MApp : forall s1 re1 s2 re2,
  exp_match s1 re1 ->
  exp_match s2 re2 ->
  exp_match (s1 ++ s2) (App re1 re2)
| MUnionL : forall s1 re1 re2,
  exp_match s1 re1 re2,
  exp_match s1 (Union re1 re2)
| MUnionR : forall s2 re2 re1,
  exp_match s2 re2,
  exp_match s2 (Union re1 re2)
| MStar0 : forall re, exp_match [] (Star re)
| MStarApp : forall s1 s2 re,
  exp_match s1 re ->
  exp_match s2 (Star re) ->
  exp_match (s1 ++ s2) (Star re).

Notation "s =~ re" := (exp_match s re) (at level 80).

Binary Trees

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

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