CIS 500: Software Foundations  Midterm I
October 12, 2021

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. Questions for just one of the tracks 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.

  To make sure, please find the questions for the other track as soon as you begin the exam and cross them out!

• 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!).

Mark the box of the track you are following.

☐ Standard  ☐ Advanced
(8 points) Put an X in the True or False box for each statement, as appropriate.

(a) This proposition is provable in Coq with no axioms:
\[ \forall (f : A \to A) (x y : A), x = y \to f x = f y. \]
\[ \square \ True \quad \square \ False \]

(b) \( \emptyset =^\sim \text{Star re} \) is provable for every \( \text{re} \). (The definition of \( =^\sim \) can be found in the “For Reference” section at the end.)
\[ \square \ True \quad \square \ False \]

(c) This proposition is provable in Coq with no axioms:
\[ \forall A (f : A \to A) (x y : \text{nat}), f x = f y \to x = y. \]
\[ \square \ True \quad \square \ False \]

(d) This proposition is provable in Coq with no axioms:
\[ \text{False} \to \text{False}. \]
\[ \square \ True \quad \square \ False \]

(e) The result of Compute (In 42 [1;2]) is False.
\[ \square \ True \quad \square \ False \]

(f) Functions defined in Coq via Fixpoint must terminate on all inputs, but functions defined with Definition need not always terminate.
\[ \square \ True \quad \square \ False \]

(g) For every property of numbers \( P : \text{nat} \to \text{Prop} \), we can construct a boolean function \( \text{testP : nat} \to \text{bool} \) such that \( \text{testP} \) reflects \( P \).
\[ \square \ True \quad \square \ False \]

(h) There exists a proposition \( P \) such that the proposition \( \neg \neg P \leftrightarrow P \) is provable (with no additional axioms).
\[ \square \ True \quad \square \ False \]
2 Standard Track Only (10 points)

(a) How many subgoals will we have after running the tactic inversion $H$?

$H: [a; b] = []$

\[ 2 = 2 \]

$\square$ Tactic fails.

$\square$ 0 (solves the goal)

$\square$ 1

$\square$ 2

$\square$ 3

(b) How many subgoals will we have after running the tactic apply $H$?

$P, Q, R: \text{Prop}$

$H: P \rightarrow Q \rightarrow R$

\[ R \]

$\square$ Tactic fails.

$\square$ 0 (solves the goal)

$\square$ 1

$\square$ 2

$\square$ 3

(c) How many subgoals will we have after running the tactic apply $H$ in $H1$?

$P, Q, R: \text{Prop}$

$H: P \rightarrow Q \rightarrow R$

$H1: P$

\[ R \]

$\square$ Tactic fails

$\square$ 0 (solves the goal)

$\square$ 1

$\square$ 2

$\square$ 3
(d) How many subgoals will we have after running the tactic induction $H$? (The definition of $\text{le}$ can be found in the “For Reference” section at the end.)

\[
\begin{align*}
n, m & : \text{nat} \\
H & : \text{lt } n \text{ m} \\
\text{---------------------} \\
\text{le } n \text{ m}
\end{align*}
\]

\begin{itemize}
  \item $\square$ Tactic fails
  \item $\square$ 0 (solves the goal)
  \item $\square$ 1
  \item $\square$ 2
  \item $\square$ 3
\end{itemize}

(e) How many subgoals will we have after running the tactic apply ($\text{le}_S n n \ (\text{le}_n n)$)?

\[
\begin{align*}
n & : \text{nat} \\
\text{---------------------} \\
\text{le } n \ (\text{S } n)
\end{align*}
\]

\begin{itemize}
  \item $\square$ Tactic fails
  \item $\square$ 0 (solves the goal)
  \item $\square$ 1
  \item $\square$ 2
  \item $\square$ 3
\end{itemize}
3 [Standard Track Only] (15 points) What is the type of each of the following Coq expressions? (Check “none of the above” if the expression is typeable but none of the given choices is its type. Check “ill-typed” if the expression does not have a type.)

(a) $4 \leq 3$
- leq
- False
- false
- Prop
- nat->nat->Prop
- ill-typed
- none of the above

(b) $\forall (A : Type) (m n : A), m = n \lor m \neq n$
- forall (A : Type) (m n : A), Prop
- forall (A : Type) A -> A -> Prop
- fun (A : Type) => fun (m n : A) => m =? n
- Prop
- True
- False
- ill-typed
- none of the above

(c) $\text{fun} (x : \text{nat}) => \text{False}$
- Prop
- nat -> Prop
- True
- False
-forall (n : nat), false
- forall (n : nat), False
- ill-typed
- none of the above
(d) forall (m : nat), m * m
   □ Prop
   □ Prop * Prop
   □ (nat,nat)
   □ False
   □ false
   □ nat -> nat
   □ fun (m : nat) => nat
   □ ill-typed
   □ none of the above

(e) beq_nat 3
   □ (nat,nat)
   □ bool
   □ Prop
   □ nat -> bool
   □ nat -> Prop
   □ ill-typed
   □ none of the above

(f) fun (P Q : Prop) => P -> Q
   □ (nat,nat)
   □ bool
   □ Prop
   □ Prop -> Prop
   □ Prop -> Prop -> Prop
   □ forall (P Q : Prop), Prop
   □ ill-typed
   □ none of the above
(g) fun (m : nat) (E : 0 <= m) => le_S 0 m E

- Prop
- nat -> Prop
- Prop -> Prop
- forall (m:nat), Prop -> Prop
- forall (m:nat), 0 <= m -> 0 <= S m
- forall (m:nat), 0 <= m -> Prop
- ill-typed
- none of the above
(a) nat -> (nat -> bool)

(b) forall (X Y : Type), list X -> list Y

(c) forall (X Y : Type), X -> (X->X->Y) -> Y

(d) forall (X Y : Type) (f : X -> Y), Y

(e) Prop -> bool

(f) In 2 [1;1;1]

(g) ev 1
(i) \( \forall n : \text{nat}, \text{ev} \; n \rightarrow \text{ev} \; (S \; (S \; n)) \)

(i) \( (\text{nat} \rightarrow \text{nat}) \rightarrow \text{nat} \)
(12 points) The higher-order function fold_left...

Fixpoint fold_left {A B} (f: B -> A -> B) (a: list A) (b: B) : B :=
  match a with
  | [] => b
  | h :: ts => fold_left f ts (f b h)
end.

... is quite versatile — in fact we can easily define many commonly used functions non-recursively, just by applying fold_left to appropriate arguments. For example this is how we can define map using fold_left:

Definition map {A B} (f: A -> B) (a: list A) : list B :=
  fold_left (fun acc e => acc ++ [f e]) a [].

Define the following functions using fold_left.

(a) Keep the elements of the input list for which the predicate \( f \) yields true.
Example: filter evenb [1;2;3;4] = [2;4]

Definition filter {A} (f: A -> bool) (a: list A) :=
fold_left

(b) From a list of pairs, return a pair of lists.
Example: unzip [(1, true); (2, false); (3, true)] = [1;2;3] [true; false; true]

Definition unzip {X Y} (l: list (X*Y)) : (list X * list Y) :=
fold_left
(c) Apply a predicate $f$ on each element of a list and return a pair of lists; if $f$ is true for a given element, put it on the left list, otherwise put it on the right list.

Example: $\text{split evenb } [1;2;3;4] = ([2;4], [1;3])$

```coq
Definition split {X} (l: list X) (f: X -> bool) : (list X * list X) :=
fold_left
```
An expression in Gallina is said to be *canonical* if it cannot be simplified. For example, these expressions are canonical:

\[ 0 \]
\[ S \ 0 \]
\[ S \ (S \ 0) \]
\[ true \]
\[ [true] \]

while these are not:

\[ 0 + 1 \]
\[ negb \ true \]
\[ [true] ++ [] \]
\[ (fun (x:nat) \Rightarrow true) \ 3 \]

Note that the type \texttt{bool} has two canonical members, while \texttt{nat} has infinitely many.

The same notion of “canonical member” also works for expressions whose types involve \texttt{Prop}. For example, given the definition of the binary \texttt{\leq} relation from the IndProp chapter:

\[
\text{Inductive \ le : nat -> nat -> Prop :=}
\quad | \ \text{le}_n (n : nat) : \leq n n
\quad | \ le_S (n m : nat) (H : \leq n m) : \leq n (S m).
\]

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


the proposition \texttt{1\leq2} has one canonical member, namely

\[ \text{le}_S \ 1 \ 1 \ (\text{le}_n \ 1) \]

while the proposition \texttt{1\leq0} is empty.

Each sub-question on the next page presents an inductively defined property \texttt{P} of natural numbers and asks you to list the canonical members of \texttt{P n} for some \texttt{n}. If \texttt{P n} has infinitely many canonical members, write “infinite.” If it has no members, write “empty.”

6.1 Define \texttt{P} as follows:

\[
\text{Inductive \ P : nat -> Prop :=}
\quad | \ A : P \ 0
\quad | \ B : P \ 1
\quad | \ C : P \ 0.
\]

What are the canonical members of \texttt{P 0}? (List all of them in the space below.)
6.2 Define $P$ as follows:

Inductive $P : \text{nat} \to \text{Prop}$ :=
| A : P 0 
| B (n : nat) : P n.

What are the canonical members of $P \ 0$?

6.3 Define $P$ as follows:

Inductive $P : \text{nat} \to \text{Prop}$ :=
| B (n:nat) (H: P n) : P (S n).

What are the canonical members of $P \ 1$?
Define \( P \) as follows:

\[
\text{Inductive } P : \text{nat} \rightarrow \text{Prop} := \\
| A : P \ 1 \\
| B (n : \text{nat}) (H : P \ (S \ n)) : P \ (S \ n).
\]

What are the canonical members of \( P \ 1 \)?

Define \( P \) as follows:

\[
\text{Inductive } P : \text{nat} \rightarrow \text{Prop} := \\
| A : P \ 1 \\
| B (n : \text{nat}) (H : n \neq n) : P \ n.
\]

What are the canonical members of \( P \ 1 \)?

Define \( P \) as follows:

\[
\text{Inductive } P : \text{nat} \rightarrow \text{Prop} := \\
| A (n : \text{nat}) (H0 : n \leq 1) : P \ n.
\]

What are the canonical members of \( P \ 1 \)?
7 (12 points) In this problem we will be working with the following definition of single-variable polynomials over the natural numbers.

```coq
Inductive Poly :=
| Var
| Const (a: nat)
| Sum (a b: Poly)
| Prod (a b: Poly).
```

The associative law for addition says that changing a subexpression of the form \( x + (y + z) \) to \((x + y) + z\) or vice versa yields an equivalent polynomial.

Your job is to complete the definition of the inductive relation `reassoc`, where `reassoc p1 p2` means that `p1` and `p2` are “equivalent modulo associativity of plus.” For example,

```coq
reassoc (Prod (Sum (Const 0)
  (Sum (Const 1) (Const 2)))
  (Const 3))
(Prod (Sum (Sum (Const 0) (Const 1))
  (Const 2))
  (Const 3)).
(* i.e., \((0 + (1 + 2)) \ast 3\) is equivalent to \(((0 + 1) + 2) \ast 3\) *)
```

We’ve given you a few of the constructors; you supply the rest.

```coq
Inductive reassoc : Poly -> Poly -> Prop :=
| refl : forall p,
  reassoc p p
| trans : forall p1 p2 p3,
  reassoc p1 p2 ->
  reassoc p2 p3 ->
  reassoc p1 p3
| sum : forall p1 p1' p2 p2',
  reassoc p1 p1' ->
  reassoc p2 p2' ->
  reassoc (Sum p1 p2) (Sum p1' p2')
```
Let's translate some English statements about polynomials into Coq theorems. First, some definitions...

An *evaluation* function for polynomials can be written as follows:

```
Fixpoint eval(p: Poly)(x: nat): nat :=
  match p with
  | Var => x
  | Const n => n
  | Sum a b => eval a x + eval b x
  | Prod a b => eval a x * eval b x
  end.
```

A polynomial is *constant* if it always yields the same result, no matter the value of the variable:

```
Definition constant (p : Poly) : Prop :=
  exists r, forall n, eval p n = r.
```

Two polynomials are *equivalent* if they yield the same result for every value of the variable:

```
Definition equiv (p1 p2 : Poly) : Prop :=
  forall n, eval p1 n = eval p2 n.
```

The *degree* of a polynomial is the highest power of the variable that appears in its “fully multiplied out” form. For example $x \cdot x + x + 2 + x \cdot x \cdot 3$ and $(x + 1) \cdot (x + 2)$ both have degree 2. Here is a definition of the *degree* function.

```
Fixpoint degree(p: Poly): nat :=
  match p with
  | Var => 1
  | Const a => 0
  | Sum a b => max (degree a) (degree b)
  | Prod a b => degree a + degree b
  end.
```

(a) Write a theorem stating that “degree-zero polynomials are constant and vice versa.” (No need to prove it—just state the theorem.)

```
Theorem deg0_constant : forall (p : Poly),
```

(b) Write a theorem stating that “Every polynomial of degree at most 1 is equivalent to one of the form $ax + b$."

```
Theorem nf : forall (p : Poly),
```
Recall the definition of In

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

end.

and the following lemma from Logic.v:

\[
\text{Lemma In_app_iff : forall } A \text{ l l'} (a:A),
\text{In a (l++l')} \leftrightarrow \text{In a l } \lor \text{In a l'}.
\]

Give a careful informal proof of the left-to-right direction of this theorem. If your proof goes by induction, make sure to state any induction hypotheses explicitly.

\[
\text{Lemma In_app_iff : forall } A \text{ l l'} (a:A),
\text{In a (l++l')} \rightarrow \text{In a l } \lor \text{In a l'}.
\]

Proof:
Recall the Fixpoint definition of list membership from the Logic chapter:

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

If we define a simple datatype of binary trees...

\[
\text{Inductive } \text{tree} \{A : \text{Type}\} : \text{Type} :=
\begin{align*}
&\text{leaf} \\
&\text{node} (\text{label} : A) (\text{ll} \text{ rr} : \text{tree } A).
\end{align*}
\]

... we can give a similar definition of “tree membership” like this:

\[
\text{Fixpoint } \text{TIn} \{A : \text{Type}\} (x : A) (t : \text{tree } A) : \text{Prop} :=
\begin{align*}
&\text{match } t \text{ with} \\
&\quad | \text{leaf } _ \Rightarrow \text{False} \\
&\quad | \text{node } _ a \text{ ll rr} \Rightarrow a = x \lor \text{TIn } x \text{ } \text{ll} \lor \text{TIn } x \text{ } \text{rr} \\
&\text{end.}
\end{align*}
\]

Next, let’s define a function squish that flattens a tree into the list of its labels:

\[
\text{Fixpoint } \text{squish} \{A : \text{Type}\} (t : \text{tree } A) : \text{list } A :=
\begin{align*}
&\text{match } t \text{ with} \\
&\quad | \text{leaf } _ \Rightarrow [] \\
&\quad | \text{node } _ a \text{ ll rr} \Rightarrow [a] ++ \text{squish } \text{ll} ++ \text{squish } \text{rr} \\
&\text{end.}
\end{align*}
\]

Now we can state a theorem saying, informally, that “squishing commutes with membership”—i.e., that a given element \(x\) is a member of a tree \(t\) iff \(x\) is a member of \(\text{squish } t\).

\[
\text{Theorem } \text{TIn_squish} : \forall A \ (x : A) \ (t : \text{tree } A), \ \text{In } x \ (\text{squish } t) \Rightarrow \text{TIn } x \ t.
\]

On the next page, give a careful informal proof of this theorem. If your proof goes by induction, make sure to state any induction hypotheses \textit{explicitly}.
Theorem TIn_squish : \( \forall A \ (x : A) \ (t : \text{tree} \ A), \)
\( \text{In} \ x \ (\text{squish} \ t) \rightarrow \text{TIn} \ x \ t. \)

Proof:
Fixpoint beq_nat(a b: nat): bool :=

  match a, b with
  | S a', S b' => beq_nat a' b'
  | 0, 0 => true
  | _, _ => false
end.

Inductive list (X:Type) : Type :=

  | nil
  | cons (x : X) (l : list X).

Fixpoint fold_left {A B} (f: B -> A -> B) (a: list A) (b: B) : B :=

  match a with
  | [] => b
  | h :: ts => fold_left f ts (f b h)
end.

Fixpoint In {A : Type} (x : A) (l : list A) : Prop :=

  match l with
  | [] => False
  | x' :: l' => x' = x / In x l'
end.

Inductive le : nat -> nat -> Prop :=

  | le_n (n : nat) : le n n
  | le_S (n m : nat) (H : le n m) : le n (S m).

Notation "n <= m" := (le n m).

Definition lt (n m: nat) := le (S n) m.

Inductive ev : nat -> Prop :=

  | ev_0 : ev 0
  | ev_SS (n : nat) (H : ev n) : ev (S (S n)).
Inductive reg_exp (T : Type) : Type :=
| EmptySet
| EmptyStr
| Char (t : T)
| App (r1 r2 : reg_exp T)
| Union (r1 r2 : reg_exp T)
| Star (r : reg_exp T).

Inductive exp_match {T} : list T -> reg_exp T -> Prop :=
| MEmpty : [] =~ EmptyStr
| MChar x : [x] =~ (Char x)
| MApp s1 re1 s2 re2
  (H1 : s1 =~ re1)
  (H2 : s2 =~ re2)
  : (s1 ++ s2) =~ (App re1 re2)
| MUnionL s1 re1 re2
  (H1 : s1 =~ re1)
  : s1 =~ (Union re1 re2)
| MUnionR re1 s2 re2
  (H2 : s2 =~ re2)
  : s2 =~ (Union re1 re2)
| MStar0 re : [] =~ (Star re)
| MStarApp s1 s2 re
  (H1 : s1 =~ re)
  (H2 : s2 =~ (Star re))
  : (s1 ++ s2) =~ (Star re)

where "s =~ re" := (exp_match s re).