CIS 500: Software Foundations

Midterm I

October 8, 2020

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.* To make sure, please look for 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
(a) The `injection` tactic takes any hypothesis of the form \( f \ x = f \ y \) and creates a hypothesis of the form \( x = y \) for any \( f \).

(b) Given propositions \( P \) and \( Q \), if \( P \rightarrow Q \) and \( Q \rightarrow P \) then \( P = Q \).

(c) Suppose our goal is \( 2 + 2 = 5 \) and our context contains the hypothesis \( H : 2 \neq 2 \). Running `discriminate H.` would complete the proof.

(d) One way to prove a proposition \( P \rightarrow Q \) in Coq is to assume \( P \) and derive a contradiction.

(e) One way to prove a proposition \( P \) in Coq is to assume \( \neg P \) and derive a contradiction.

(f) Suppose we have a hypothesis \( H : 1 = 2 \). Running `injection H.` will give us the assumption \( 0 = 1 \).

(g) Coq’s termination checker will reject every `Fixpoint` definition that does not terminate on every input.
(h) Coq’s termination checker will accept *every* `Fixpoint` definition that does terminate on all inputs.

(i) A function like `fun x => x+1` is simply a special kind of proposition in Coq.
2. [Standard Track Only] (13 points)

Write the type of each of the following Coq expressions (write “ill typed” if an expression does not have a type).

(a) fun n => match n with S m => true | 0 => false end
(b) if true then True else False
(c) if True then true else false
(d) nat -> bool
(e) True -> False
(f) fun (H : False) => 5
(g) False -> (forall n, n = S n)
(h) fun (X : Type) (f : X -> X -> X) (x : X) => f x x
(i) (fun (X : Type) (f : X -> X -> X) (x : X) => f x x) nat plus
(j) fun (n : nat) (b : bool) => n <> b
(k) fun (H : forall (x : nat), x = x) => H :: nil
(l) fun x => if false then x else [0]
(m) fun x => plus x
3  (14 points)

For each of the following types, either give a term of this type or write “uninhabited” if there are no terms of this type.

(a) $\forall (X \ Y : \text{Type}), (X -> Y) -> X -> Y$

(b) $\text{list False}$

(c) $\forall (X \ Y \ Z : \text{Type}), ((X * Y) -> Z) -> (X -> Y -> Z)$

(d) $(\text{fun} \ (X : \text{Type}) => \text{bool}) \text{ nat}$

(e) $\text{nat} -> (\forall (X : \text{Type}), (X -> X) -> X -> X)$

(f) $\text{False} -> \text{bool}$

(g) $\text{True} -> \text{Prop}$
An expression in Gallina is said to be canonical if it cannot be simplified. For example, these expressions are canonical

\[
\begin{align*}
0 \\
S\ 0 \\
S\ (S\ 0) \\
true \\
[\text{true}]
\end{align*}
\]

while these are not:

\[
\begin{align*}
0\ +\ 1 \\
negb\ true \\
[\text{true}]\ ++\ [] \\
(fun\ (x:\text{nat}) \Rightarrow true)\ 3
\end{align*}
\]

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

The same definition works for expressions whose types involve \texttt{Prop}. For example, given the definition of $\leq$ from the IndProp chapter,

\[
\begin{align*}
\text{Inductive}\ le:\ \text{nat} \rightarrow \text{nat} \rightarrow \text{Prop} := \\
| le_n (n:\text{nat}) : le\ n\ n \\
| le_S (n\ m:\text{nat}) (H:\ le\ n\ m) : le\ n\ (S\ m).
\end{align*}
\]

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

the proposition $1 \leq 2$ has one canonical member, namely

$le_S\ 1\ 1\ (le_n\ 1)$

while the proposition $1 \leq 0$ is empty.

For each of the following (parameterized) propositions, list all the canonical members of all concrete instances of the proposition, together with their types — i.e., for every canonical expression $e$ such that $e : P\ n$ for some number $|n|$. you should write "$e : P\ n$" — or else write "infinite" if there are infinitely many. If the proposition has no members, write “empty.”

For example, for the proposition

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

you would write

$A\ 0\ (le_S\ 0\ 0\ (le_n\ 0)) : P\ 0$

$A\ 1\ (le_n\ 1) : P\ 1$

and for the proposition

\[
\begin{align*}
\text{Inductive}\ P:\ \text{nat} \rightarrow \text{Prop} := \\
| A\ (n:\text{nat}) (H0 : 5 \leq n) : P\ n.
\end{align*}
\]

you would write “infinite.”
(a) \textbf{Inductive} \ P \ : \ \text{nat} \to \text{Prop} := \\
| \ A : \ P \ 1 \\
| \ B \ (n : \text{nat}) \ (H : \ n \not= \ n) : \ P \ n.

(b) \textbf{Inductive} \ P \ : \ \text{nat} \to \text{Prop} := \\
| \ A : \ P \ 1 \\
| \ B : \ P \ 3 \\
| \ C \ (n : \text{nat}) \ (H0 : \ n = 2) \ (H1 : \ P \ n) : \ P \ (S \ n).

(c) \textbf{Inductive} \ P \ : \ \text{nat} \to \text{Prop} := \\
| \ A : \ P \ 1 \\
| \ B \ (n:\text{nat}) \ (H : \ P \ (S \ n)) : \ P \ (S \ n).

(d) \textbf{Inductive} \ P \ : \ \text{nat} \to \text{Prop} := \\
| \ A : \ P \ 1 \\
| \ B \ (n:\text{nat}) \ (H : \ P \ (S \ n)) : \ P \ n.

(e) \textbf{Inductive} \ P \ : \ \text{nat} \to \text{Prop} := \\
| \ B \ (n:\text{nat}) \ (H : \ P \ n) : \ P \ (S \ n).
Recall the polymorphic “fold” function over lists.

\[
\text{Fixpoint foldr \{X Y\} (f : X \to Y \to Y) (l : list X) (b : Y) : Y :=}
\]
\[
\begin{align*}
\text{match l with} \\
\text{\quad |} \text{[]} \Rightarrow b \\
\text{\quad |} h :: t \Rightarrow f h (\text{foldr} f t b) \\
\text{end.}
\end{align*}
\]

We’ve renamed this function “foldr” here because it folds “from the right” to the left. Another variant of the same idea is to “fold from the left”:

\[
\text{Fixpoint foldl \{X Y\} (f : Y \to X \to Y) (l : list X) (b : Y) : Y :=}
\]
\[
\begin{align*}
\text{match l with} \\
\text{\quad |} \text{[]} \Rightarrow b \\
\text{\quad |} h :: t \Rightarrow \text{foldl} f t (f b h) \\
\text{end.}
\end{align*}
\]

Both versions apply their function argument acc sequentially to all elements of a list, but they do so in different ways.

(a) Suppose we are given types \(X\) and \(Y\), a function \(f : X \to Y \to X\), and four values \(a : X\) and \(b\ c\ d : Y\). Simplify the following expression: \(\text{foldl} f [b;c;d] a\).

(b) Suppose we are given types \(X\) and \(Y\), a function \(f : X \to Y \to Y\), and four values \(a\ b\ c : X\) and \(d : Y\). Simplify the following expression: \(\text{foldr} f [a;b;c] d\).

(c) Using the above functions, define two functions that compute the sum of the elements of a list \(\text{nat}\).

\[
\text{Definition sumlistl (l : list \text{nat}) : \text{nat} :=}
\]
\[
\text{foldl (* fill in arguments here: *)}
\]
\[
\text{Definition sumlistr (l : list \text{nat}) : \text{nat} :=}
\]
\[
\text{foldr (* fill in arguments here: *)}
\]

(d) As the previous question suggests, there are some functions \(f\) on which \(\text{foldl}\) and \(\text{foldr}\) behave identically.

Give another example (besides \texttt{plus}) of such a function.

(e) On the other hand, this is not true for \textit{all} functions \(f\). Give an example of a type \(X\), a function \(f : X \to X \to X\), a value \(\text{init} : X\), and a list \(l : \text{list} X\) such that \(\text{foldl} f l \text{ init}\) and \(\text{foldr} f l \text{ init}\) yield different results.

\[
\bullet X =
\]
(f) State sufficient conditions on $f$ such that given any $\text{init} : X$ and $l : \text{list } X$, 
$\text{foldl } f \ l \ \text{init} = \text{foldr } f \ l \ \text{init}.$

Express your conditions as predicates of type $\forall X, (X \rightarrow X \rightarrow X) \rightarrow \text{Prop}.$
Propositions A and B are *logically equivalent* to each other when proposition A holds if and only if proposition B holds.

If we have a *list* of statements and we wish to show that they are all logically equivalent to each other, it is often easier to write a *cyclic proof*. Rather than considering all pairs of statements pairwise, we can prove a single circular chain (cycle) of implications that connects them all.

Define an inductive proposition that holds true for a list of implications form a cyclic chain of implications.

Complete the definition of the inductive proposition below.

```plaintext
Inductive cyclic_implication : list Prop -> Prop :=
  Impl_Cons : forall (P Q R : Prop) l, (P -> Q) -> (R -> P) ->
                 cyclic_implication (Q :: l ++ [R]) ->
                 cyclic_implication (P :: Q :: l ++ [R])
```

Here is one way of defining the “sortedness” property for lists of natural numbers:

\[\text{Fixpoint } \text{sorted1} (l : \text{list nat}) :=\]
\[\text{match } l \text{ with}\]
\[\text{| } \text{[] } \Rightarrow \text{True}\]
\[\text{| } x :: l' \Rightarrow \text{(forall } y, \text{ In } y l' \rightarrow x \leq y) \land \text{sorted1 } l'\]
\[\text{end.}\]

Rewrite this property as an Inductive definition, keeping the same informal intuition (“a list is \text{sorted1} if it is empty or if its head element is less than or equal to every element in the tail and the tail is \text{sorted1}”).

\[\text{Inductive } \text{sorted1} : \text{list nat } \rightarrow \text{ Prop} :=\]
Here is the definition of sortedness again:

\[
\text{Fixpoint sorted1} \ (l : \text{list nat}) := \\
\quad \text{match} \ l \ \text{with} \\
\quad \quad | \ [] \Rightarrow \text{True} \\
\quad \quad | \ x :: l' \Rightarrow (\forall y, \text{In} y l' \Rightarrow x \leq y) \land \text{sorted1} l'
\end{align*}
\]

And here is an alternate definition based on a slightly different intuition:

\[
\text{Inductive sorted2} : \text{list nat} \rightarrow \text{Prop} := \\
\quad \text{SEmpty} : \text{sorted2} [] \\
\quad \text{SSing} : \forall x, \text{sorted2} [x] \\
\quad \text{SCons} : \forall x1 x2 l', x1 \leq x2 \Rightarrow \text{sorted2} (x2::l') \Rightarrow \text{sorted2} (x1::x2::l').
\]

Write a careful informal proof of the following fact. If your proof uses induction, make sure to state the induction hypothesis explicitly and rigorously.

\[
\text{Theorem t12} : \forall l, \text{sorted1} l \Rightarrow \text{sorted2} l.
\]
Recall the definitions of the `reg_exp` datatype and the `exp_match` relation.

```ocaml
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).

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

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).
```

Using `reg_exp_match`, we can derive a natural equivalence relation on regular expressions:

```ocaml
Definition equiv {T : Type} (a b : reg_exp T) :=
  forall s,
  s =~ a <-> s =~ b.
```

In this problem you may refer the following theorem (proved in the homework) without proof.

```ocaml
Theorem star_app : forall T (s1 s2 : list T) (e : reg_exp T),
  s1 =~ Star e ->
  s2 =~ Star e ->
  (s1 ++ s2) =~ Star e.
```

Also, for the sake of brevity you may use the notation `a*` to mean `Star a`.

Write a careful informal proof of the following proposition. When dealing with inductive cases, explicitly state what the inductive hypotheses are.

```ocaml
Theorem aistarstar : forall T (a : reg_exp T),
equiv (Star a) (Star (Star a)).
```