Solutions
(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:
\[
\forall (f : A \to A) \ (x \ y : A), \ x = y \to f \ x = f \ y.
\]
- ⊥ True □ False

(b) ⊤ =~ Star re is provable for every re. (The definition of =~ can be found in the “For Reference” section at the end.)
- ⊥ True □ False

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

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

(e) The result of Compute (In 42 [1;2]) is False.
- □ True ⊥ False

(f) Functions defined in Coq via Fixpoint must terminate on all inputs, but functions defined with Definition need not always terminate.
- □ True ⊥ False

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

(h) There exists a proposition \( P \) such that the proposition \( \neg P \leftrightarrow P \) is provable (with no additional axioms).
- ⊥ True □ False
(a) How many subgoals will we have after running the tactic inversion $H$?

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

\[ 2 = 2 \]

☐ Tactic fails.

☒ 0 (solves the goal)

☐ 1

☐ 2

☐ 3

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

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

$H: P \rightarrow Q \rightarrow R$

\[ R \]

☐ Tactic fails.

☐ 0 (solves the goal)

☐ 1

☒ 2

☐ 3

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

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

$H: P \rightarrow Q \rightarrow R$

$H1: P$

\[ R \]

☐ Tactic fails

☐ 0 (solves the goal)

☐ 1

☒ 2

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

n, m: nat
H: lt n m
-------------------
le n m

□ Tactic fails
□ 0 (solves the goal)
□ 1
☑ 2
□ 3

(e) How many subgoals will we have after running the tactic apply (le_S n n (le_n n))? 

n: nat
-------------------
le n (S n)

□ Tactic fails
☑ 0 (solves the goal)
□ 1
□ 2
□ 3
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) \(\forall (A : \text{Type}) \ (m \ n : A), m = n \lor m \not= n\)

- □ \(\forall (A : \text{Type}) \ (m \ n : A), \text{Prop}\)
- □ \(\forall (A : \text{Type}) \ A \to A \to \text{Prop}\)
- □ fun \(A : \text{Type}) \Rightarrow\text{fun}(m \ n : A) \Rightarrow m =? n\)
- ☒ Prop
- □ True
- □ False
- □ ill-typed
- □ none of the above

(c) \(\text{fun} \ (x : \text{nat}) \Rightarrow \text{False}\)

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

(e) \text{beq_nat} 3
- \(\Box\) (nat,nat)
- \(\Box\) bool
- \(\Box\) Prop
  - \(\times\) nat --> bool
- \(\Box\) nat --> Prop
- \(\Box\) ill-typed
- \(\Box\) none of the above

(f) \text{fun} (P Q : \text{Prop}) => P --> Q
- \(\Box\) (nat,nat)
- \(\Box\) bool
- \(\Box\) Prop
- \(\Box\) Prop --> Prop
  - \(\times\) Prop --> Prop --> Prop
- \(\Box\) forall (P Q : Prop), Prop
- \(\Box\) ill-typed
- \(\Box\) 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
(15 points) For each of the types below, write a Coq expression that has that type, or else write “uninhabited” if there are no such expressions.

(a) \(\text{nat} \to (\text{nat} \to \text{bool})\)
   
   \textit{Answer: Example: leb}

(b) \(\forall (X \ Y : \text{Type}), \text{list } X \to \text{list } Y\)

   \textit{Answer: Example: fun X Y (l : list X) => []}

(c) \(\forall (X \ Y : \text{Type}), X \to (X \to X \to Y) \to Y\)

   \textit{Answer: Example: fun X Y (a : X) (f : X \to X \to Y) => f a a}

(d) \(\forall (X \ Y : \text{Type}) \ (f : X \to Y), Y\)

   \textit{Answer: Uninhabited}

(e) \(\text{Prop} \to \text{bool}\)

   \textit{Answer: Example: fun (P : Prop) => true}

(f) \(\text{In } 2 [1;1;1]\)

   \textit{Answer: Uninhabited}

(g) \(\text{ev } 1\)

   \textit{Answer: Uninhabited}

(h) \(\forall n : \text{nat}, \text{ev } n \to \text{ev } (S \ (S \ n))\)

   \textit{Answer: Example: ev_SS}

(i) \((\text{nat} \to \text{nat}) \to \text{nat}\)

   \textit{Answer: Example: fun f => (f 0)}
The higher-order function `fold_left`...

```coq
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`:

```coq
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]`

```coq
Definition filter {A} (f: A -> bool) (a: list A) :=
  fold_left (fun acc e => if f e then acc ++ [e] else acc) a [].
```

(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]`

```coq
Definition unzip {X Y} (l: list (X*Y)) : (list X * list Y) :=
  fold_left (fun acc e =>
    match acc with
    | (a, b) => (a ++ [fst e], b ++ [snd e])
    end)
  l ([],[]).
```
(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: \( \text{split evenb } [1;2;3;4] = ([2;4], [1;3]) \)

\[
\text{Definition split } \{X\} (l: \text{list } X) (f: X \rightarrow \text{bool}) : (\text{list } X \times \text{list } X) :=
\]

\[
\text{fold_left}
\]

\[
\begin{array}{l}
(\text{fun acc e =>} \\
\quad \text{match acc with} \\
\quad \mid (a, b) \Rightarrow \text{if } f \ e \ \text{then } (a ++ \ [e], b) \ \text{else } (a, b ++ \ [e]) \\
\quad \end{array}
\]

\[
1 ([], []).\]
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) => true) 3
```

Note that the type `bool` has two canonical members, while `nat` has infinitely many.

The same notion of “canonical member” also works for expressions whose types involve `Prop`. For example, given the definition of the binary <= relation from the IndProp chapter:

```
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).

the proposition `1<=2` has one canonical member, namely

```
le_S 1 1 (le_n 1)
```

while the proposition `1<=0` is empty.

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

6.1 Define `P` as follows:

```
Inductive P : nat -> Prop :=
| A : P 0
| B : P 1
| C : P 0.
```

What are the canonical members of `P 0`? (List all of them in the space below.)

Check A : P 0.
Check C : P 0.
6.2 Define $P$ as follows:

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

What are the canonical members of $P \text{ 0}$?

Check $A : P \text{ 0}$.
Check $B \text{ 0} : P \text{ 0}$.

6.3 Define $P$ as follows:

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

What are the canonical members of $P \text{ 1}$?

(* Empty! *)
Define \( P \) as follows:

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

What are the canonical members of \( P \text{~} 1 \)?

(* Infinite! *)
Check \( A : P \text{~} 1 \).
Check \( B \text{~} 0 A : P \text{~} 1 \).
Check \( B \text{~} 0 (B \text{~} 0 A) : P \text{~} 1 \).
Check \( B \text{~} 0 (B \text{~} 0 (B \text{~} 0 A)) : P \text{~} 1 \).
Check \( B \text{~} 0 (B \text{~} 0 (B \text{~} 0 (B \text{~} 0 A))) : P \text{~} 1 \).

Define \( P \) as follows:

\[
\text{Inductive } P : \text{nat} \to \text{Prop} := \\
| A : P \text{~} 1 \\
| B (n:\text{nat}) (H : n \not= n) : P n.
\]

What are the canonical members of \( P \text{~} 1 \)?
Check \( A : P \text{~} 1 \).

Define \( P \) as follows:

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

What are the canonical members of \( P \text{~} 1 \)?
Check \( A \text{~} 1 \text{~} (\text{le}_n \text{~} 1) : P \text{~} 1 \).
(12 points) In this problem we will be working with the following definition of single-variable polynomials over the natural numbers.

\[\text{Inductive } \text{Poly} :=
| \text{Var} \\
| \text{Const } (a: \text{nat}) \\
| \text{Sum } (a \ b: \text{Poly}) \\
| \text{Prod } (a \ b: \text{Poly}).\]

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 \(\text{reassoc}\), where \(\text{reassoc } p1 \ p2\) means that \(p1\) and \(p2\) are "equivalent modulo associativity of plus." For example,

\[
\text{reassoc } \text{Prod } (\text{Sum } (\text{Const } 0) \\
(\text{Const } 3)) \\
(\text{Prod } (\text{Sum } (\text{Const } 0) (\text{Const } 1)) \\
(\text{Const } 2)) \\
(\text{Const } 3)).
\]

(* i.e., \((0 + (1 + 2)) \times 3\) is equivalent to \(((0 + 1) + 2) \times 3\) *)

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

\[\text{Inductive } \text{reassoc } : \text{Poly } \rightarrow \text{Poly } \rightarrow \text{Prop} :=
| \text{refl } : \forall p, \\
\text{reassoc } p \ p \\
| \text{trans } : \forall p1 \ p2 \ p3, \\
\text{reassoc } p1 \ p2 \rightarrow \\
\text{reassoc } p2 \ p3 \rightarrow \\
\text{reassoc } p1 \ p3 \\
| \text{sum } : \forall p1 \ p1' \ p2 \ p2', \\
\text{reassoc } p1 \ p1' \rightarrow \\
\text{reassoc } p2 \ p2' \rightarrow \\
\text{reassoc } (\text{Sum } p1 \ p2) (\text{Sum } p1' \ p2') \\
| \text{prod } : \forall p1 \ p1' \ p2 \ p2', \\
\text{reassoc } p1 \ p1' \rightarrow \\
\text{reassoc } p2 \ p2' \rightarrow \\
\text{reassoc } (\text{Prod } p1 \ p2) (\text{Prod } p1' \ p2') \\
| \text{assoc } : \forall p1 \ p2 \ p3, \\
\text{reassoc } (\text{Sum } p1 (\text{Sum } p2 \ p3)) (\text{Sum } (\text{Sum } p1 \ p2) \ p3) \\
| \text{symm } : \forall p1 \ p2, \\
\text{reassoc } p1 \ p2 \rightarrow \\
\text{reassoc } p2 \ p1.\]
Let’s translate some English statements about polynomials into Coq theorems. First, some definitions...

An evaluation function for polynomials can be written as follows:

```coq
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:

```coq
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:

```coq
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.

```coq
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.)

```coq
Theorem deg0_constant : forall (p : Poly),
  degree p = 0 <-> constant p.
```

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

```coq
Theorem nf : forall (p : Poly),
  degree p <= 1
  <-> exists a b, equiv p (Sum (Prod (Const a) Var) (Const b)).
```
Recall the definition of $\text{In}$

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

$$\begin{array}{c}
\text{match } l \text{ with} \\
| [] => \text{False} \\
| x' :: l' => x' = x \lor \text{In } x \text{ l'}
\end{array}$$

and the following lemma from Logic.v:

$$\text{Lemma In_app_iff : forall } A \ l \ l' \ (a:A), \text{ In a (l++l') <-> In a l \lor 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 \ l \ l' \ (a:A), \text{ In a (l++l') -> In a l \lor In a l'}. \$$

$$\text{Proof: By induction on the list } l.$$

Base case: $l = []$. In this case, $l++l' = l'$, so

$$\text{In a (l++l') = In a l'},$$

and the result is immediate.

Induction case: $l = h::t$, with induction hypothesis

$$\text{IH = In a (t++l') -> In a l \lor In a l'}.$$

By the definition of $\text{In}$, we know

$$\text{In a (l++l')}$$

i.e., $\text{In a ((h::t)++l')}$$

i.e., $\text{In a (h::(t++l'))}$

i.e., $a=h \lor \text{In a (t++l')}$

Suppose $a=h$. Then

$$a=h \lor \text{In a t}$$

i.e., $\text{In a (h::t)}$

i.e., $\text{In a l}$

and the result is immediate.

The other possibility is $\text{In a (t++l')}$, and the result is again immediate.
Recall the Fixpoint definition of list membership from the Logic chapter:

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

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

\[
\text{Inductive } \text{tree} \{A : \text{Type}\} : \text{Type} := \\
\text{| } \text{leaf} \\
\text{| } \text{node} (\text{label} : A) (\text{ll} \text{ rr} : \text{tree} \ A).
\]

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

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

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

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

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}. 

16
Theorem TIn_squish : forall A (x : A) (t : tree A),
In x (squish t) -> TIn x t.

Proof:

By induction on t.

- Base case: t = leaf. Then squish t is [] by definition, and 
  TIn A x t = In A x (squish t) = False.
  The result is immediate.

- Induction case: We are given 
  t = node a ll rr 
  IH1: In A x (squish ll) -> TIn A x ll 
  IH2: In A x (squish rr) -> TIn A x rr

By the definition of squish,

  squish t = [a] ++ (squish ll ++ squish rr).

Reason as follows:

\[
\begin{align*}
& \text{In A x (squish t)} \\
& \iff \text{In A x [a] \lor In A x ll \lor In A x rr.} \\
& \text{(by In_app_iff, twice)} \\
& \implies \text{In A x [a] \lor TIn A x (squish ll) \lor TIn A x (squish rr).} \\
& \text{(by IH1 and IH2)} \\
& \implies x = a \lor \text{False} \lor TIn A x (squish ll) \lor TIn A x (squish rr). \\
& \text{(by the definition of In)} \\
& \iff x = a \lor \text{TIn A x (squish ll) \lor TIn A x (squish rr).} \\
& \text{(by the definition of TIn A x (node a ll rr)}, \\
& \text{In A x (squish t) -> x = TIn A x t}
\end{align*}
\]

as required.
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).