Solutions
Write the type of each of the following Coq expressions (write “ill typed” if an expression does not have a type). The definitions of le and Sn_le_Sm__n_le_m can be found on page 1 of the appendix, for reference.

(a) \(\forall (n : \text{nat}), n \leq n\)

\[\text{Answer: Prop}\]

(b) \(\text{fun} (n : \text{nat}) \Rightarrow n \leq n\)

\[\text{Answer: } \text{nat} \rightarrow \text{Prop}\]

(c) \(\text{fun} (P : \text{Prop}) \Rightarrow P \rightarrow \text{False}\)

\[\text{Answer: } \text{Prop} \rightarrow \text{Prop}\]

(d) \(\text{le} 5\)

\[\text{Answer: } \text{nat} \rightarrow \text{Prop}\]

(e) \(\text{fun} (n : \text{nat}) \Rightarrow [n] :: [1;2;3]\)

\[\text{Answer: ill typed}\]

(f) \(\text{fun} (n m : \text{nat}) (H : S n \leq S m) \Rightarrow \text{Sn.le_Sm__n.le_m} n m H\)

\[\text{Answer: } \forall (n m : \text{nat}), S n \leq S m \Rightarrow n \leq m\]
2. [Standard Track Only] From Types to Terms (10 points)

For each of the types below, write a Coq expression that has that type or write “empty” if the type has no inhabitants. The definition of the $\leq$ relation can be found on page 1 of the appendix, for reference.

(a) $\text{bool} \rightarrow \text{Prop}$
   
   \text{Answer: } \text{fun} \ (b : \text{bool}) \Rightarrow \text{False}

(b) $\forall (b : \text{bool}) \rightarrow \text{bool}$

\text{Answer: } \text{negb}

(c) $\forall (P : \text{Prop}), P \rightarrow \text{False}$

\text{Answer: } \text{Empty}

(d) $(\text{bool} \rightarrow \text{nat} \rightarrow \text{bool}) \rightarrow \text{bool}$

\text{Answer: } \text{fun } f \Rightarrow (f \ \text{true} \ \text{0})

(e) $\forall (X : \text{Type}), \text{list} \ X \rightarrow X$

\text{Answer: } \text{Empty}
3  [Standard Track Only] Inductively Defined Relations (11 points)

Suppose we are given the following definition of binary trees.

```
Inductive tree : Type :=
  | Leaf (n : nat)
  | Node (n : nat) (left : tree) (right : tree).
```

An in-order traversal of a binary tree is the list of node and leaf labels computed as follows:

- If the tree is a leaf, the in-order traversal is a singleton list of the leaf’s label.
- If the tree is a node, the in-order traversal contains...
  - first the labels from an in-order traversal of the left subtree,
  - then the label of the node,
  - then the labels from an in-order traversal of the right subtree

For example, the in-order traversal of the tree:

```
Node 1
  (Node 2
    (Leaf 4)
    (Leaf 5))
(Leaf 3)
```

is the following list:

\[4; 2; 5; 1; 3\]

3.1 Complete the inductive definition of a binary relation `inorder` that relates a binary tree with its in-order traversal list.

```
Inductive inorder : tree -> list nat -> Prop :=
  | In_Leaf :
    forall n,
    inorder (Leaf n) [n]
  | In_Node :
    forall n ltree rtree l1 l2,
    inorder ltree l1 ->
    inorder rtree l2 ->
    inorder (Node n ltree rtree) (l1 ++ [n] ++ l2).
```

3.2 Using the rules of your inductive definition from above, write a proof term for the following claim about the example tree shown above. (Write the whole term explicitly, without using any underscores.)
Definition inorder_test :

inorder

(Node 1 (Node 2 (Leaf 4) (Leaf 5)) (Leaf 3))

[4; 2; 5; 1; 3]

:= In_Node 1 (Node 2 (Leaf 4) (Leaf 5)) (Leaf 3) [4; 2; 5] [3]

(In_Node 2 (Leaf 4) (Leaf 5) [4] [5]

(In_Leaf 4) (In_Leaf 5))

(In_Leaf 3).
4  [Standard Track Only] Functional Programming (11 points)

The height of a binary tree is the length of the longest path from the root to a leaf, plus one. For example, the height of the example tree on page 3 is 3.

4.1 Define a function `height` that computes the height of a binary tree.

```coq
Fixpoint height (tr : tree) : nat :=
  match tr with
  | Leaf _ => 1
  | Node _ ltree rtree =>
    if height ltree <? height rtree
      then 1 + (height rtree)
      else 1 + (height ltree)
  end.
```

or...

```coq
Fixpoint height' (tr : tree) : nat :=
  match tr with
  | Leaf _ => 1
  | Node _ ltree rtree => 1 + max (height' ltree) (height' rtree)
  end.
```

4.2 Suppose we define the notion of a `path` in a tree as follows:

```coq
Inductive tree_step :=
  | go_left
  | go_right.

Definition path := list tree_step.
```

That is, a path is a list of steps to either right or left, starting from the root of the tree. For example the path `[go_left; go_right]` says to start at the root, go to the left subtree, then go to the left subtree’s right subtree and stop. Following this path in the example tree on page 3 would lead us to `Leaf 5`.

Define a function that returns the longest path in a given tree from the root to a leaf—that is, it should calculate the list of steps to reach the leaf at the end of the longest path. If there is more than one longest path, it should return the leftmost one. For example, it should return `[go_left; go_left]` for the example tree above. You may find the `height` function from above and/or the `max` function on page 1 of the appendix useful.

```coq
Fixpoint longest_path (tr : tree) : path :=
  match tr with
  | Leaf _ => []
  | Node _ ltree rtree =>
    if height ltree <? height rtree
      then go_right :: (longest_path rtree)
      else go_left :: (longest_path ltree)
  end.
```
Loop Invariants for Hoare Logic (18 points)

For each program below, give an invariant for the WHILE loop that will allow us to prove the stated Hoare triple. (Note that we are silently extending Imp to contain all the usual boolean comparison operators—<, =<, >, >=, to make the programs easier to understand). The Coq variables m and n stand for arbitrary values of type nat.

(a)  \[\{ X = m \}\]
    \[\text{WHILE } X >= 0 \text{ DO}\]
    \[X ::= X + 1\]
    \[\{\text{Invariant goes here}\}\]
    \[\text{END}\]
    \[\{ X = m \}\]

Invariant = \(X >= m\) or True. Anything that is implied by the precondition and preserved by the loop.

(b) The min and max functions are defined on page 1 of the appendix.

\[\{ X = m /\ Y = n \}\]
\[\text{Z1 ::= 0;}\]
\[\text{WHILE } X > 0 \text{ && } Y > 0 \text{ DO}\]
\[\text{Z1 ::= Z1 + 1;}\]
\[X ::= X - 1;\]
\[Y ::= Y - 1;\]
\[\{\text{Invariant goes here}\}\]
\[\text{END;}\]
\[\text{Z2 ::= Z1 + X + Y}\]
\[\{ Z1 = \text{min } m \text{ n } /\ Z2 = \text{max } m \text{ n } \}\]

Invariant = \(Z1 + \text{min } X \text{ Y = min } m \text{ n } /\ Z1 + \text{max } X \text{ Y = max } m \text{ n}\)

(c)  \[\{ X = m \}\]
    \[\text{WHILE } X > 0 \text{ DO}\]
    \[X ::= 1\]
    \[\{\text{Invariant goes here}\}\]
    \[\text{END}\]
    \[\{ X = 0 \}\]

Invariant = \(X = m \wedge X = 1\) True

(d)  \[\{ X = m \}\]
    \[Z ::= 0;\]
    \[Y ::= 0;\]
    \[\text{WHILE } (X > 0 \text{ || } Y > 0) \text{ DO}\]
    \[\text{IF } Y == 0 \text{ THEN}\]
    \[Z ::= Z + 1;\]
    \[Y ::= X - 1;\]
    \[X ::= 0\]
ELSE
   Z ::= Z + 1;
   X ::= Y - 1;
   Y ::= 0
FI
{{ Invariant goes here }}
END
{{ Z = m }}

Invariant = (Y = 0 /\ Z + X = m) \ (/ (X = 0 /\ Z + Y = m)
Theorem while_no_terminate:

forall (b : bexp) (c : com) (st : state),
(forall st1 st2, st1 = [\ c \] => st2 -> beval st1 b = beval st2 b) ->
beval st b = true ->
\(\neg (\exists st', st = [\ \text{WHILE} \ b \ \text{DO} \ c \ \text{END} \ ] => st')\).

To prove this theorem, we first assume, for a contradiction, that there does exist such an \(st'\) for some given \(b\), \(c\), and \(st\). After some rearrangement\(^1\) we find that, from these global assumptions

\[
\begin{align*}
& b : bexp \\
& c : com \\
& Hpres : \forall st1 st2, \\
& st1 = [\ c \] => st2 -> \\
& \text{beval} st1 b = \text{beval} st2 b
\end{align*}
\]

we must show the following proposition:

\[
\begin{align*}
& \forall (cc : com) (st st' : state), \\
& st = [\ cc \] => st' -> \\
& cc = \text{WHILE} b \ \text{DO} c \ \text{END} -> \\
& \text{beval} st b = \text{true} -> \\
& \text{False}
\end{align*}
\]

Carefully complete an informal proof of this proposition on the next page. If you use induction, remember to clearly state what you are performing induction on, and give the explicit induction hypothesis explicitly in each case where you use it.

Proof: By induction on a derivation of \(st = [\ cc \] => st'\). The only possible cases are \(E_{\text{WhileFalse}}\) and \(E_{\text{WhileTrue}}\), because we know that \(cc = \text{WHILE} b \ \text{DO} c \ \text{END}\).

- For the \(E_{\text{WhileFalse}}\) case, we know that \(\text{beval} st b = \text{false}\), which contradicts with \(\text{beval} st b = \text{true}\) so we are done.

- For the \(E_{\text{WhileTrue}}\) case, we know that:

\[
\begin{align*}
& st'' : state \\
& H' : \text{beval} st b = \text{true} \\
& \text{Hev1} : st = [\ c \] => st'' \\
& \text{Hev2} : st'' = [\ \text{WHILE} b \ \text{DO} c \ \text{END} \ ] => st'
\end{align*}
\]

and we have the following two inductive hypotheses:

\(^1\)Namely: turning \(\neg (\exists x, P)\) into \(\forall x, P \rightarrow \text{False}\), introducing \(cc\) to generalize the proposition about how the command evaluates, and shuffling some quantifiers around.
IH : \text{WHILE } b \text{ DO } c \text{ END} = \text{WHILE } b \text{ DO } c \text{ END} \to \text{beval st'' } b = \text{true} \to \text{False}

IH' : c = \text{WHILE } b \text{ DO } c \text{ END} \to \text{beval st } b = \text{true} \to \text{False}

By using the induction hypothesis IH we now need to show that \text{WHILE } b \text{ DO } c \text{ END} = \text{WHILE } b \text{ DO } c \text{ END} (which follows from reflexivity) and \text{beval st'' } b = \text{true}.

\text{beval st'' } b = \text{true} \text{ can be proven using Hpres, which states that if } st = [c] \Rightarrow st'' \text{ then } \text{beval st } b = \text{beval st'' } b. \text{ Since we know } st = [c] \Rightarrow st'' \text{ by Hev1, we can conclude that } \text{beval st } b = \text{beval st'' } b, \text{ and since } \text{beval st } b = \text{true} \text{ by H', we can conclude that } \text{beval st'' } b = \text{true} \text{ as well.}
Reduction Order in the STLC (20 points)

Consider the simply typed lambda-calculus with booleans, enriched with an extra term called `diverge`:

(* types as before... *)
Inductive ty : Type :=
| Bool : ty
| Arrow : ty -> ty -> ty
| Ty_Prod : ty -> ty -> ty.

(* terms as before... *)
Inductive tm : Type :=
| tm_var : string -> tm
| tm_app : tm -> tm -> tm
| tm_abs : string -> ty -> tm -> tm
| tm_true : tm
| tm_false : tm
| tm_if : tm -> tm -> tm -> tm
(* ...plus one new one: *)
| tm_diverge : tm.

Notation "'diverge'" := tm_diverge (in custom stlc at level 0).

The term `diverge` represents an infinite loop; this is achieved by making `diverge` step to itself:

Inductive step : tm -> tm -> Prop :=
(* ... step rules as before, plus: *)
| ST_Diverge :
<<diverge>> --> <<diverge>>

Since `diverge` never reduces to a value, it is safe to give it any type at all:

Inductive has_type : context -> tm -> ty -> Prop :=
(* ... typing rules as before, plus: *)
| T_Diverge : forall Gamma T,
Gamma |- <<diverge>> \in T

Note that both CBV and CBN re deterministic reduction strategies: CBV performs all possible reductions on the argument to an application before substituting it into the function’s body, while CBN does no reduction in the argument before doing the substitution.

Continued on the next page...
Two terms $t_1$ and $t_2$ are said to coterminate if either (1) both reach normal forms after a finite number of steps (not necessarily the same normal form or after the same number of steps) or (2) both diverge (keep reducing and never reach normal forms). For each of the following subproblems, mark whether or not the two given terms coterminate for all possible subterms $t$ of the appropriate type:

$t_1 = (\lambda x: \text{Bool}, \text{if } x \text{ then diverge else false}) \; t$

$t_2 = (\lambda x: \text{Bool}, \text{if } (\text{if } x \text{ then false else true}) \text{ then true else diverge}) \; t$

- ☒ Coterminate
- □ Don't coterminate

**Counterexample (if they don’t coterminate, give a value for $t$):**

$t_1 = (\lambda x: \text{Bool}, \text{if } (\text{if } x \text{ then true else diverge}) \text{ then true else diverge}) \; t$

$t_2 = (\lambda x: \text{Bool}, \text{diverge}) \; t$

□ Coterminate
- ☒ Don't coterminate

**Counterexample (if they don’t coterminate, give a value for $t$):** $t = \text{true}$

$t_1 = (\lambda f: \text{Bool->Bool}, f \text{ true}) \; t$

$t_2 = (\lambda f: \text{Bool->Bool}, \text{true}) \; t$

□ Coterminate
- ☒ Don't coterminate

**Counterexample (if they don’t coterminate, give a value for $t$):** $t = (\lambda y: \text{Bool}, \text{diverge})$
The *Software Foundations* text gives a call by value (or CBV) reduction order to STLC. This means that whenever we perform function application, the arguments to the function are always fully reduced to values before substitution, even if the function ultimately does not use its argument.

There is another commonly discussed reduction order, called call by name (or CBN) reduction, where function application does not force arguments to be values, instead substituting them into the function's body directly. For instance, consider this term:

\[(\texttt{y:Bool}, \texttt{y}) ((\texttt{x:Bool}, \text{if } \texttt{x} \text{ then false else true}) \text{ true})\]

Under call by name, this steps to

\[(\texttt{x:Bool}, \text{if } \texttt{x} \text{ then false else true}) \text{ true}\]

because we substitute \((\texttt{x:Bool}, \text{if } \texttt{x} \text{ then false else true}) \text{ true})\) for \(y\) in \((\texttt{y:Bool}, \texttt{y})\).

By contrast, under call by value it steps to

\[(\texttt{y:Bool}, \texttt{y}) (\text{if } \texttt{true} \text{ then false else true})\]

because we step \((\texttt{x:Bool}, \text{if } \texttt{x} \text{ then false else true}) \text{ true})\) before we substitute.

Complete the \texttt{cbnstep} relation below so that it uses CBN reduction for function applications. (Note that \texttt{-n->} is notation for \texttt{cbnstep}.)

Reserved Notation "t1 '-n->' t2" (at level 40).

Inductive cbnstep : tm -> tm -> Prop :=
(* rules for conditionals and diverge as before: *)
| CBN_ST_IfTrue : forall t1 t2,
  <<if true then t1 else t2>> -n-> t1
| CBN_ST_IfFalse : forall t1 t2,
  <<if false then t1 else t2>> -n-> t2
| CBN_ST_If : forall t1 t1' t2 t3,
  t1 -n-> t1' ->
  <<if t1 then t2 else t3>> -n-> <<if t1' then t2 else t3>>
| CBN_ST_Diverge :
  <<diverge>> -n-> <<diverge>>

(* Add rules as needed below: *)

| CBN_ST_App1 : forall t1 t1' t2,
  t1 -n-> t1' ->
  <<t1 t2>> -n-> <<t1' t2>>
| CBN_ST_AppAbs : forall x T t12 t2,
  <<(\texttt{x:T}, t12) t2>> -n-> [x:=t2]t12

For each of the following, check the appropriate box to indicate whether or not \(t\) terminates to a value under each reduction order. If it terminates, give the final value that it reduces to under that reduction order.

\(t = (\texttt{x:Bool}, (\texttt{y:Bool}, x)) ((\texttt{x:Bool}, x) \text{ diverge})\)
(a) □ Terminates under CBV...
   ☒ Diverges under CBV
   ...and evaluates to:

(b) ☒ Terminates under CBN...
   □ Diverges under CBN
   ...and evaluates to: \( y: \text{Bool}, ((x: \text{Bool}, x) \text{ diverge}) \)

\[
t = (\langle f: \text{Bool} \to \text{Bool}, (y: \text{Bool}, f) \rangle (\langle x: \text{Bool}, (z: \text{Bool}, x) \rangle \text{ true})
\]

(a) ☒ Terminates under CBV...
   □ Diverges under CBV
   ...and evaluates to: \( y: \text{Bool}, (z: \text{Bool}, \text{ true}) \)

(b) ☒ Terminates under CBN...
   □ Diverges under CBN
   ...and evaluates to: \( y: \text{Bool}, ((x: \text{Bool}, (z: \text{Bool}, x)) \text{ true}) \)

\[
t = (\langle f: \text{Bool} \to \text{Bool}, (x: \text{Bool}, f \text{ true}) \rangle (y: \text{Bool}, \text{ if } y \text{ then false else true})) \text{ diverge}
\]

(a) □ Terminates under CBV...
   ☒ Diverges under CBV
   ...and evaluates to:

(b) ☒ Terminates under CBN...
   □ Diverges under CBN
   ...and evaluates to: \( \text{false} \)
CBN reduction can sometimes be more efficient than CBV, in the sense that it won’t evaluate arguments that aren’t used by a function. For instance:

\[
(\lambda x: \text{Bool}, (\lambda z: \text{Bool}, \text{if } z \text{ then false else true}) \text{ true}) ((\lambda y: \text{Bool}, y) \text{ true})
\]

-n-> \[
(\lambda z: \text{Bool}, \text{if } z \text{ then false else true}) \text{ true}
\]

However, with CBV this will take more steps to fully reduce:

\[
(\lambda x: \text{Bool}, (\lambda z: \text{Bool}, \text{if } z \text{ then false else true}) \text{ true}) ((\lambda y: \text{Bool}, y) \text{ true})
\]

--> \[
(\lambda x: \text{Bool}, (\lambda z: \text{Bool}, \text{if } z \text{ then false else true}) \text{ true}) \text{ true}
\]

--> \[
(\lambda z: \text{Bool}, \text{if } z \text{ then false else true}) \text{ true}
\]

Is CBN reduction always more efficient than the CBV order—i.e., does it always take the same or fewer steps to reach to a value? If so, briefly explain why. If not, give a counterexample. (You do not need to explicitly show the counterexample evaluating—just give a term that takes more steps with CBN than with CBV.)

**Answer:** No, this is not always more efficient, you might end up reducing an argument multiple times if it’s duplicated in the function body. For instance, consider the term:

\[
(\lambda x: T \rightarrow T, x (x v)) ((\lambda z: (T \rightarrow T) \rightarrow (T \rightarrow T), z) (\lambda y: T, y))
\]

Under *call by value* reduction we have:

\[
(\lambda x: T \rightarrow T, x (x v)) ((\lambda z: (T \rightarrow T) \rightarrow (T \rightarrow T), z) (\lambda y: T, y))
\]

\[
(\lambda x: T \rightarrow T, x (x v)) (\lambda y: T, y)
\]

\[
(\lambda y: T, y) (\lambda y: T, y) v
\]

\[
(\lambda y: T, y) v
\]

But under *call by name* reduction we have:

\[
(\lambda x: T \rightarrow T, x (x v)) ((\lambda z: (T \rightarrow T) \rightarrow (T \rightarrow T), z) (\lambda y: T, y))
\]

\[
((\lambda z: (T \rightarrow T) \rightarrow (T \rightarrow T), z) (\lambda y: T, y)) ((\lambda z: (T \rightarrow T) \rightarrow (T \rightarrow T), z) (\lambda y: T, y)) v
\]

\[
(\lambda y: T, y) ((\lambda z: (T \rightarrow T) \rightarrow (T \rightarrow T), z) (\lambda y: T, y)) v
\]

\[
((\lambda z: (T \rightarrow T) \rightarrow (T \rightarrow T), z) (\lambda y: T, y)) v
\]

\[
(\lambda y: T, y) v
\]

Which takes one more step than CBV did.
8 [Advanced Track Only] Progress for Call-By-Name (12 points)

Fill in the missing cases of the following informal proof of the progress theorem for the CBN reduction relation that you wrote on page 12. Make sure to state the induction hypothesis explicitly where appropriate. The definition of value is repeated on page 5 of the appendix, for reference (it is the same as always).

\[
\text{Theorem cbn_progress :}
\]
\[
\text{forall t T, empty |- t \in T -> value t \lor (exists t', t -n-> t').}
\]

Proof: By induction on t.

Case \( t = \text{true, false, or a conditional} \)
(Skip these.)

Case \( t = x \)
Can’t happen (t must be typeable in the empty context).

Case \( t = \text{dive}\)erge
Since \( t \) steps to itself, we just need to choose \( t' = t \) and we are done.

Case \( t = \lambda x:T, t1 \)
In this case we know
\[
\begin{align*}
\bullet & \ T = T1 \rightarrow T2 \\
\bullet & \ t = \text{\(\langle\langle\lambda x:T1, \text{body}\rangle\\rightarrow\text{\(\langle\langle\lambda x:T1, \text{body}\rangle\\rightarrow\text{\(\langle\rangle)}\}} \) for some T1 and some body} \\
\bullet & \ (X |- T1) |- body \in T2
\end{align*}
\]
and we just need to show that \( t \) either is a value or can take a step. Since abstractions are values, this case is trivial.
Case \( t = t_1 t_2 \)

We have two IHs in this case:

\[
\begin{align*}
\forall T_1, \emptyset |- t_1 \in T_1 \to \text{value } t_1 \lor (\exists t_1' : \text{tm}, t_1 \rightarrow t_1') \\
\forall T_2, \emptyset |- t_2 \in T_2 \to \text{value } t_2 \lor (\exists t_2' : \text{tm}, t_2 \rightarrow t_2')
\end{align*}
\]

By inverting the typing hypothesis, we get

\[
\begin{align*}
\emptyset |- t_1 \in T_2 \to T \\
\emptyset |- t_2 \in T_2
\end{align*}
\]

and hence:

\[
\begin{align*}
\text{value } t_1 \lor (\exists t_1' : \text{tm}, t_1 \rightarrow t_1') \\
\text{value } t_2 \lor (\exists t_2' : \text{tm}, t_2 \rightarrow t_2')
\end{align*}
\]

We want to show:

\[
\text{value } \langle t_1 t_2 \rangle \lor (\exists t' : \text{tm}, \langle t_1 t_2 \rangle \rightarrow t')
\]

In particular, we will show that \( \langle t_1 t_2 \rangle \) can step. We first consider whether or not \( t_1 \) is a value or can step by case analysis on the first of our induction hypotheses.

- \( t_1 \) is a value:
  
  Since \( t_1 \) is a value and has a function type in the empty context, we know (by inverting the typing relation) that there exists some \( x \) and \( \text{body} \) such that \( t_1 = \langle \lambda x : T_2, \text{body} \rangle \), and we know
  
  \( \langle \lambda x : T_2, \text{body} \rangle \rightarrow t_2 \rightarrow [x := t_2] \text{body} \)
  
  by CBN_ST_AppAbs.

- \( t_1 \) can step. If \( t_1 \rightarrow t_1' \) then we know
  
  \( \langle t_1 t_2 \rangle \rightarrow \langle t_1' t_2 \rangle \)
  
  by CBN_ST_App1
Inductively Defined Relations / Free Variables in the STLC (12 points)

This problem concerns the STLC with booleans and conditionals (page 4 in the appendix). A closed term in this language is one that has no free variables. Formally, the textbook defines closed terms with respect to an “appears free in” relation (see page 5 in the appendix).

In this question we explore an alternate definition of closure: Instead of defining what it means for a variable to appear free in a term and then saying that a closed term is one in which no variable appears free, we define what it means for a term to be closed except for a certain list of variables that may appear free. Formally, a term $t$ is said to be closed under a list $env$ if all the variables that appear free in $t$ are in $env$. A closed term, then, is simply one that is closed under the empty list of variables. For instance:

Theorem closed_under_1 : forall (x y z : string),
closed_under [x;y;z] x.

Theorem closed_under_2 : forall (x y z : string),
closed_under [] <<\x:Bool, x>>.

Theorem closed_under_3 : forall (x : string),
~ closed_under [] x.

Complete the following definition of closed_under. Give just the cases for variables, applications, and abstractions and omit the cases for if, true, or false. Do not use appears_free_in or the old definition of closure from Software Foundations in your answer. Do feel free to use the standard In relation for list membership (In x l means “x appears in l”).

Inductive closed_under : list string -> tm -> Prop :=

| cu_var : forall (x : string) (env : list string),
  In x env -> closed_under env <<x>>

| cu_app : forall t1 t2 env,
  closed_under env t1 ->
  closed_under env t2 ->
  closed_under env <<t1 t2>>

| cu_abs : forall x T body env,
  closed_under (x :: env) body ->
  closed_under env <<\x:T, body>>

| cu_if : forall cond ifbody elsebody env,
  closed_under env cond ->
  closed_under env ifbody ->
  closed_under env elsebody ->
  closed_under env <<if cond then ifbody else elsebody>>

| cu_true : forall env,
  closed_under env <<true>>

| cu_false : forall env,
  closed_under env <<false>>.
Write a careful informal proof of the following theorem, which relates `appears_free_in` to the new definition of `closed_under_env` from the previous question. Intuitively, the theorem states that, if all variables that appear free in `t` are in the list `env`, then `t` is closed under `env`.

Theorem `appears_free_in_closed_under`:
```latex
forall t env,
(foreall x, appears_free_in x t -> In x env) ->
closed_under env t.
```

Again, give just the cases for variables, applications, and abstractions. If you use induction, remember to clearly state what you are performing induction on, as well as your induction hypotheses for each inductive case.

Proof: We will prove this theorem by induction on `t`.

- `t = s`: where `s` is a variable.
  In this case we have the assumption:
  ```latex
  forall x, appears_free_in x s -> In x env
  ```
  And we have no inductive hypotheses.
  we want to show that this implies `closed_under_env env s`.
  We can use our assumption to show that `In s env`, as `appears_free_in s s` holds by `afi_var s`. Knowing `In s env` we can derive `closed_under_env env s` by `cu_var`.

- `t = <<\s:T, body>>` for some `s`, `T`, and `body`.
  In this case we have the assumption:
  ```latex
  forall x, appears_free_in x <<\s:T, body>> -> In x env
  ```
  And the inductive hypothesis
  ```latex
  forall env, (forall x, appears_free_in x body -> In x env) -> closed_under env body
  ```
  And we want to show that this implies `closed_under_env env <<\s:T, body>>`. We could show this using `cu_abs` as long as we can show `closed_under (s :: env) body`.
  We can derive `closed_under (s :: env) body` from our IH as long as we can show that:
  ```latex
  forall x, appears_free_in x body -> In x (s :: env)
  ```
  Which follows from the assumption that `forall x, appears_free_in x <<\s:T, body>> -> In x env`, since this implies that `x` is either free in the body, or `x=s` (and clearly `In s (s :: env)`).

- `t = <<t1 t2>>` for some `t1` and `t2`.
  In this case we have the assumption:
forall x, appears_free_in x <<t1 t2>> -> In x env

And the inductive hypotheses

- IH1: forall env, (forall x, appears_free_in x t1 -> In x env) -> closed_under env t1
- IH2: forall env, (forall x, appears_free_in x t2 -> In x env) -> closed_under env t2

And we want to show that closed_under env <<t1 t2>>. We can show this via cu_app if we can prove closed_under_env t1 and closed_under_env t2. Both of these cases are nearly identical, so we will focus on the closed_under_env t1 case.

We see that closed_under_env t1 is the conclusion of IH1, and thus we just need to show:

forall x, appears_free_in x t1 -> In x env

Given appears_free_in x t1 as an assumption we can derive appears_free_in x <<t1 t2>> by afi_app1, and then we can use our assumptions to show that In x env.
**Subtyping** (12 points)
Consider the simply typed lambda-calculus with booleans, products, and subtyping, (see pages 4 and 5 in the appendix).
Observe that the type $\text{Top} \times \text{Top}$ has exactly one “proper supertype” (that is, one supertype other than itself)—namely Top. On the other hand, the type $(\text{Top} \times \text{Top}) \times (\text{Top} \times \text{Top})$ has exactly four proper supertypes:

$(\text{Top} \times \text{Top}) \times \text{Top}$
$\text{Top} \times (\text{Top} \times \text{Top})$
$\text{Top} \times \text{Top}$
$\text{Top}$

11.1 Using just the Top type and the product type constructor, complete the three missing rows in the following table:

<table>
<thead>
<tr>
<th>N</th>
<th>Example of a type with exactly N proper supertypes</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Top</td>
</tr>
<tr>
<td>1</td>
<td>Top \times Top</td>
</tr>
<tr>
<td>2</td>
<td>(Top \times Top) \times Top</td>
</tr>
<tr>
<td>3</td>
<td>((Top \times Top) \times Top) \times Top</td>
</tr>
<tr>
<td>4</td>
<td>(Top\timesTop) \times (Top\timesTop)</td>
</tr>
</tbody>
</table>

11.2 Using just the Top and Bool types and the arrow type constructor, complete the following table:

<table>
<thead>
<tr>
<th>N</th>
<th>Example of a type with exactly N proper supertypes</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Top</td>
</tr>
<tr>
<td>1</td>
<td>Bool</td>
</tr>
<tr>
<td>2</td>
<td>Bool-&gt;Bool</td>
</tr>
<tr>
<td>3</td>
<td>Bool-&gt;Bool-&gt;Bool</td>
</tr>
<tr>
<td>4</td>
<td>Bool-&gt;Bool-&gt;Bool-&gt;Bool</td>
</tr>
</tbody>
</table>
11.3 Using just \texttt{Top} and products, give an example of a type with exactly two proper subtypes (or write “None” if no such type exists).

Answer: None

11.4 Using just \texttt{Top}, \texttt{Bool}, and arrows, give an example of a type with exactly two proper subtypes (or write “None” if no such type exists).

Answer: (\texttt{Bool}→\texttt{Bool})→\texttt{Bool}
Subtyping and Typechecking (14 points)

In this problem we consider the simply typed lambda calculus with booleans, pairs, and subtyping. The syntax, operational semantics, and typing rules are given in the Appendix.

Recall that this language already includes the type $\text{Top}$, which is a supertype of all other types, as indicated by this subtyping rule:

\[
\begin{array}{c}
\hline
S < : \text{Top} \\
\end{array}
\]

In this problem we consider the implications of adding a new type $\text{Bot}$ (for “bottom”), which is a subtype of all others:

\[
\begin{array}{c}
\hline
\text{Bot} < : S \\
\end{array}
\]

Just as when we added $\text{Top}$, we leave the operational semantics and the typing rules unchanged.

(a) For each of the following lemmas, indicate whether it is provable with the addition of $\text{Bot}$ as described above:

- Lemma sub_inversion_Bot : forall $U$, $U < : \text{Bot}$ -> $U = \text{Bot}$.
  
  ❑ Provable    □ Not provable

- Lemma sub_inversion_Bool : forall $U$, $U < : \text{Bool}$ -> $U = \text{Bool}$.
  
  □ Provable    ❑ Not provable

- Lemma canonical_forms_of_Bool : forall $s$,
  
  empty $\vdash s \in \text{Bool}$ ->
  
  value $s$ ->
  
  ($s = \text{ttrue}$ $\lor$ $s = \text{tfalse}$).
  
  ❑ Provable    □ Not provable

(b) Recall that a term is closed if it has no free variables. Are there any closed values of type $\text{Bot}$? That is, can you find a value $v$ such that:

\[
\text{empty} \vdash v : \text{Bot}
\]

If so, give an example. If not, briefly explain.

Answer: No. By induction on the possible typing derivation: there are no base-case values of type $\text{Bot}$, and subsumption would require a $T$ such that $T < : \text{Bot}$, but (again by induction) in that case $T = \text{Bot}$. 
(c) For each pair of types $T$ and $S$ given below, indicate whether $T <: S$, $S <: T$, or $T$ and $S$ are incomparable (that is, not related by $<:$).

- $T = \text{Bool} \rightarrow \text{Top}$  
  $S = \text{Bot} \rightarrow \text{Bot}$

  ✗ $T <: S$  
  ✓ $S <: T$  
  ✗ incomparable

- $T = (\text{Bot} \rightarrow \text{Top}) \rightarrow \text{Bool}$  
  $S = (\text{Top} \rightarrow \text{Bot}) \rightarrow \text{Top}$

  ✗ $T <: S$  
  ✗ $S <: T$  
  ✗ incomparable

- $T = \text{Bool} \rightarrow (\text{Top} \ast \text{Bot})$  
  $S = \text{Bot} \rightarrow (\text{Bot} \ast \text{Top})$

  ✗ $T <: S$  
  ✗ $S <: T$  
  ✗ incomparable

(d) Consider the following program:

```plaintext
empty ⊢ (\x:T. x x) : T \rightarrow \text{Bot}
```

Which of the following types $T$ allow the above program to be well-typed? That is, for which of the following choices of $T$ does there exists a typing derivation with the conclusion above?

- $T = \text{Bool} \rightarrow \text{Bool}$
- $T = \text{Bot}$
- $T = \text{Top}$
- $T = \text{Top} \rightarrow \text{Bot}$
- $T = (\text{Bot} \rightarrow \text{Bool}) \rightarrow \text{Bot}$
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

**Numeric Comparison**

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

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

Theorem \( \text{Sn_le_Sm__n_le_m} : \forall n m, S n \leq S m \rightarrow n \leq m. \)

**Minimum and Maximum**

- Definition min (a : nat) (b : nat) :=
  \[
  \text{if } a \leq? b \text{ then } a \text{ else } b.
  \]

- Definition max (a : nat) (b : nat) :=
  \[
  \text{if } a \leq? b \text{ then } b \text{ else } a.
  \]

**Imp**

Syntax:

\[
\text{Inductive } \text{aexp} : \text{Type} := \\
\mid \text{ANum} (n : \text{nat}) \\
\mid \text{APlus} (a1 a2 : \text{aexp}) \\
\mid \text{AMinus} (a1 a2 : \text{aexp}) \\
\mid \text{AMult} (a1 a2 : \text{aexp}).
\]

\[
\text{Inductive } \text{bexp} : \text{Type} := \\
\mid \text{BTrue} \\
\mid \text{BFalse} \\
\mid \text{BEq} (a1 a2 : \text{aexp}) \\
\mid \text{BLe} (a1 a2 : \text{aexp}) \\
\mid \text{BNot} (b : \text{bexp}) \\
\mid \text{BAnd} (b1 b2 : \text{bexp}).
\]

\[
\text{Inductive } \text{com} : \text{Type} := \\
\mid \text{CSkip} \\
\mid \text{CAss} (x : \text{string}) (a : \text{aexp}) \\
\mid \text{CSeq} (c1 c2 : \text{com}) \\
\mid \text{CIf} (b : \text{bexp}) (c1 c2 : \text{com}) \\
\mid \text{CWhile} (b : \text{bexp}) (c : \text{com}).
\]
Notations:

Notation "x + y" := (APlus x y) (at level 50, left associativity) : imp_scope.
Notation "x - y" := (AMinus x y) (at level 50, left associativity) : imp_scope.
Notation "x * y" := (AMult x y) (at level 40, left associativity) : imp_scope.
Notation "x <= y" := (BLe x y) (at level 70, no associativity) : imp_scope.
Notation "x = y" := (BEq x y) (at level 70, no associativity) : imp_scope.
Notation "x && y" := (BAnd x y) (at level 40, left associativity) : imp_scope.
Notation "'~' b" := (BNot b) (at level 75, right associativity) : imp_scope.
Notation "'SKIP'" :=
    CSkip : imp_scope.
Notation "x ::= 'a" :=
    (CAss x a) (at level 60) : imp_scope.
Notation "c1 ;; c2" :=
    (CSeq c1 c2) (at level 80, right associativity) : imp_scope.
Notation "'WHILE' b 'DO' c 'END'" :=
    (CWhile b c) (at level 80, right associativity) : imp_scope.
Notation "'TEST' c1 'THEN' c2 'ELSE' c3 'FI'" :=
    (CIf c1 c2 c3) (at level 80, right associativity) : imp_scope.

Evaluation:

Definition state := total_map nat.

Fixpoint aeval (st : state) (a : aexp) : nat :=
  match a with
  | ANum n => n
  | AId x => st x
  | APlus a1 a2 => (aeval st a1) + (aeval st a2)
  | AMinus a1 a2 => (aeval st a1) - (aeval st a2)
  | AMult a1 a2 => (aeval st a1) * (aeval st a2)
  end.

Fixpoint beval (st : state) (b : bexp) : bool :=
  match b with
  | BTrue => true
  | BFalse => false
  | BEq a1 a2 => (aeval st a1) =? (aeval st a2)
  | BLe a1 a2 => (aeval st a1) <=? (aeval st a2)
  | BNot b1 => negb (beval st b1)
  | BAnd b1 b2 => andb (beval st b1) (beval st b2)
  end.
Command evaluation:

\[
\begin{align*}
\text{Inductive } & \text{ceval : com} \to \text{state} \to \text{state} \to \text{Prop} := \\
| \text{E_Skip} & : \forall \text{st}, \\
& \quad \text{st} = [\text{SKIP}] \Rightarrow \text{st} \\
| \text{E_Ass} & : \forall \text{st} \ a1 \ n \ x, \\
& \quad \text{aeval} \text{st} a1 = n \rightarrow \\
& \quad \text{st} = [x := a1] \Rightarrow (x \mapsto n \ ; \text{st}) \\
| \text{E_Seq} & : \forall \text{c1 c2 st st'}, \\
& \quad \text{st} = [\text{c1}] \Rightarrow \text{st'} \rightarrow \\
& \quad \text{st}' = [\text{c2}] \Rightarrow \text{st''} \rightarrow \\
& \quad \text{st} = [\text{c1} ; ; \text{c2}] \Rightarrow \text{st''} \\
| \text{E_IfTrue} & : \forall \text{st st' b c1 c2}, \\
& \quad \text{beval} \text{st} b = \text{true} \rightarrow \\
& \quad \text{st} = [\text{c1}] \Rightarrow \text{st'} \rightarrow \\
& \quad \text{st} = [\text{TEST b THEN c1 ELSE c2 FI}] \Rightarrow \text{st'} \\
| \text{E_IfFalse} & : \forall \text{st st' b c1 c2}, \\
& \quad \text{beval} \text{st} b = \text{false} \rightarrow \\
& \quad \text{st} = [\text{c2}] \Rightarrow \text{st'} \rightarrow \\
& \quad \text{st} = [\text{TEST b THEN c1 ELSE c2 FI}] \Rightarrow \text{st'} \\
| \text{E_WhileFalse} & : \forall \text{b st c}, \\
& \quad \text{beval} \text{st} b = \text{false} \rightarrow \\
& \quad \text{st} = [\text{WHILE b DO c END}] \Rightarrow \text{st} \\
| \text{E_WhileTrue} & : \forall \text{st st' st'' b c}, \\
& \quad \text{beval} \text{st} b = \text{true} \rightarrow \\
& \quad \text{st} = [\text{c}] \Rightarrow \text{st'} \rightarrow \\
& \quad \text{st}' = [\text{WHILE b DO c END}] \Rightarrow \text{st''} \rightarrow \\
& \quad \text{st} = [\text{WHILE b DO c END}] \Rightarrow \text{st''} \\
\end{align*}
\]

where "\text{st} = [\text{c}] \Rightarrow \text{st}''" := (\text{ceval} \text{c} \text{st} \text{st}'').

Problem 6: Assumptions and Goal

Assumptions:

\[
\begin{align*}
\text{b} & : \text{bexp} \\
\text{c} & : \text{com} \\
\text{Hpres} & : \forall \text{st1 st2}, \text{st1} = [\text{c}] \Rightarrow \text{st2} \rightarrow \text{beval} \text{st1} \text{b} = \text{beval} \text{st2} \text{b}
\end{align*}
\]

Goal:

\[
\begin{align*}
\forall (\text{cc} : \text{com}) (\text{st st'} : \text{state}), \\
& \text{st} = [\text{cc}] \Rightarrow \text{st'} \rightarrow \\
& \text{cc} = \text{WHILE b DO c END} \rightarrow \\
& \text{beval} \text{st} \text{b} = \text{true} \rightarrow \\
& \text{False}
\end{align*}
\]
STLC with booleans

Syntax

\[
\begin{align*}
T & ::= \text{Bool} \\
& | T \rightarrow T \\
& | t \rightarrow T \\
& | \forall x : T, t \\
& | \text{true} \\
& | \text{false} \\
& | \text{if } t \text{ then } t \text{ else } t
\end{align*}
\]

Small-step operational semantics

\[
\begin{align*}
\text{value } v \\
\text{------------------------} \quad \text{(ST_AppAbs)} \\
(\forall x : T, t) \quad v \rightarrow [x:=v]t
\end{align*}
\]

\[
\begin{align*}
t1 \rightarrow t1' \\
\text{------------------------} \quad \text{(ST_App1)} \\
t1 \quad t2 \rightarrow t2' \\
\text{------------------------} \quad \text{(ST_App2)} \\
\text{--------------------} \quad \text{(ST_IfTrue)} \\
\text{--------------------} \quad \text{(ST_IfFalse)} \\
\text{--------------------} \quad \text{(ST_If)}
\end{align*}
\]

Typing

\[
\begin{align*}
\Gamma, x = T & \quad \text{(T_Var)} \\
\Gamma, x : T11 \vdash t12 \in T12 & \quad \text{(T_Abs)} \\
\Gamma \vdash x \in T & \quad \text{(T_Var)} \\
\Gamma \vdash t1 \in T12 \rightarrow T12 & \quad \text{(T_True)} \\
\Gamma \vdash t2 \in T11 & \quad \text{(T_Var)} \\
\Gamma \vdash \text{false} \in \text{Bool} & \quad \text{(T_False)} \\
\end{align*}
\]

(Continued on next page.)
The “Appears Free In” Relation

\[
\text{Inductive appears_free_in : string -> tm -> Prop :=}
\]

| afi_var : forall x, 
| appears_free_in x <<x>>
| afi_app1 : forall x t1 t2, 
| appears_free_in x t1 -> 
| appears_free_in x <<t1 t2>>
| afi_app2 : forall x t1 t2, 
| appears_free_in x t2 -> 
| appears_free_in x <<t1 t2>>
| afi_abs : forall x y T11 t12, 
| y <> x -> 
| appears_free_in x t12 -> 
| appears_free_in x <<y:T11, t12>>
| afi_if1 : forall x t1 t2 t3, 
| appears_free_in x t1 -> 
| appears_free_in x <<if t1 then t2 else t3>>
| afi_if2 : forall x t1 t2 t3, 
| appears_free_in x t2 -> 
| appears_free_in x <<if t1 then t2 else t3>>
| afi_if3 : forall x t1 t2 t3, 
| appears_free_in x t3 -> 
| appears_free_in x <<if t1 then t2 else t3>>.

Definition old_closed (t:tm) := 
forall x, ~ appears_free_in x t.

Hint Constructors appears_free_in.

Values for STLC with Bools

\[
\text{Inductive value : tm -> Prop :=}
\]

| v_abs : forall x T t, 
| value <<\x:T, t>>
| v_true : 
| value <<true>>
| v_false : 
| value <<false>>.

Properties of STLC

Theorem preservation : forall t t’ T, 
empty |- t \in T -> 
(t --> t’) -> 
empty |- t’ \in T.

(Continued on next page.)
Theorem progress: forall t T,
empty \vdash t \in T ->
value t \not/ exists t', t --> t'.

STLC with products

Extend the STLC with product types, terms, projections, and pair values:

\[ T ::= \ldots \]
\[ t ::= \ldots \]
\[ v ::= \ldots \]
\[ \mid T * T \]
\[ \mid (t,t) \]
\[ \mid t.fst \]
\[ \mid t.snd \]

Small-step operational semantics (added to STLC rules)

\[ \begin{array}{ll}
    t1 --> t1' & \quad \text{(ST_Pair1)} \quad (v1, t2) --> (v1', t2)
    (t1, t2) --> (t1', t2) & \quad \text{(ST_Pair2)}
  \end{array} \]

\[ \begin{array}{ll}
    t1 --> t1' & \quad \text{(ST_Fst1)} \quad (v1, v2).fst --> v1
    t1.fst --> t1'.fst & \quad \text{(ST_FstPair)}
  \end{array} \]

\[ \begin{array}{ll}
    t1 --> t1' & \quad \text{(ST_Snd1)} \quad (v1, v2).snd --> v2
    t1.snd --> t1'.snd & \quad \text{(ST_SndPair)}
  \end{array} \]

Typing (added to STLC rules)

\[ \begin{array}{ll}
    \Gamma \vdash t1 \in T1 & \quad \Gamma \vdash t2 \in T2
    \Gamma \vdash (t1,t2) \in T1*T2 & \quad \text{(T_Pair)}
  \end{array} \]

\[ \begin{array}{ll}
    \Gamma \vdash t \in T1*T2 & \quad \Gamma \vdash t \in T1*T2
    \Gamma \vdash t.fst \in T1 & \quad \Gamma \vdash t.snd \in T2 & \quad \text{(T_Fst)} \quad \text{(T_Snd)}
  \end{array} \]
STLC with Booleans, Products and Subtyping

Extend the language from pages 4 to 6 with the type Top (terms and values remain unchanged):

\[
T ::= \ldots
| \text{Top}
\]

Add these rules that characterize the subtyping relation:

\[
\begin{align*}
S <: U & \quad U <: T \quad \quad \text{(S_Trans)} & \\ 
S <: T & \quad T <: T \quad \quad \text{(S_Refl)} & \\ 
S <: \text{Top} & \quad \quad \text{(S_Top)}
\end{align*}
\]

\[
\begin{align*}
S1 <: T1 & \quad S2 <: T2 \quad \quad \text{(S_Prod)} & \\ 
T1 <: S1 & \quad S2 <: T2 \quad \quad \text{(S_Arrow)}
\end{align*}
\]

And add this to the typing relation:

\[
\begin{align*}
\Gamma |- t \in S & \quad S <: T \quad \quad \text{(T_Sub)} \quad \\ 
\Gamma |- t \in T
\end{align*}
\]