Full name or WPE-I id (printed):

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 WPE-I id or 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 (mark Advanced if you are taking the exam as a WPE-I).

☐ Standard  ☐ Advanced
Write the type of each of the following Coq expressions (write “ill typed” if an expression does not have a type). The definitions of $\text{le}$ and $\text{Sn\_le\_Sm\_\_n\_le\_m}$ can be found on page 1 of the appendix, for reference.

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

(b) $\text{fun (n : nat) => n \leq n}$

(c) $\text{fun (P : Prop) => P \rightarrow \text{False}}$

(d) $\text{le 5}$

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

(f) $\text{fun (n m : nat) (H : S n \leq S m) => Sn\_le\_Sm\_\_n\_le\_m n m H}$
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) bool $\rightarrow$ Prop

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

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

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

(e) $\forall (X : \text{Type}), \text{list } X \rightarrow X$
3 [Standard Track Only] Inductively Defined Relations (11 points)

Suppose we are given the following definition of binary trees.

\[
\text{Inductive } \text{tree : Type :=} \\
\mid \text{Leaf (n : nat)} \\
\mid \text{Node (n : nat) (left : tree) (right : tree).}
\]

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:

\[
\text{Node 1} \\
\quad \text{(Node 2)} \\
\quad \quad \text{(Leaf 4)} \\
\quad \quad \text{(Leaf 5))} \\
\quad \text{(Leaf 3)} \\
\]

is the following list:

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

Continued on the next page...
Complete the inductive definition of a binary relation \texttt{inorder} that relates a binary tree with its in-order traversal list.

\texttt{Inductive inorder : tree -> list nat -> Prop :=}

\hspace{1cm} | In Leaf : \]

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.)

\texttt{Definition inorder_test :
  inorder
  (Node 1 (Node 2 (Leaf 4) (Leaf 5)) (Leaf 3))
  [4; 2; 5; 1; 3]
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.

\[
\text{Fixpoint height (tr : tree) : nat :=}
\]

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

\[
\text{Inductive tree_step :=}
\]
\[
| \text{go_left} \]
\[
| \text{go_right}. \]

\[
\text{Definition path := list tree_step.}
\]

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 \([\text{go_left; go_right}]\) 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 \([\text{go_left; go_left}]\) 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.

\[
\text{Fixpoint longest_path (tr : tree) : path :=}
\]
5 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 \}\}
    WHILE X >= 0 DO
    X ::= X + 1
    \{\{ Invariant goes here \}\}
    END
    \{\{ X = m \}\}

Invariant =

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

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

Invariant =

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

Invariant =

6
(d)  \{\{ X = m \}\}
    \begin{align*}
    Z & ::= 0; \\
    Y & ::= 0; \\
    \text{WHILE } (X > 0 \text{ || } Y > 0) \text{ DO} \\
    & \quad \text{IF } Y == 0 \text{ THEN} \\
    & \quad \quad Z ::= Z + 1; \\
    & \quad \quad Y ::= X - 1; \\
    & \quad \quad X ::= 0 \\
    & \quad \quad \text{ELSE} \\
    & \quad \quad Z ::= Z + 1; \\
    & \quad \quad X ::= Y - 1; \\
    & \quad \quad Y ::= 0 \\
    & \quad \text{FI} \\
    & \quad \text{INVARIANT goes here} \\
    & \text{END} \\
    & \{\{ Z = m \}\}
\end{align*}

Invariant =
The following theorem says that, if \( b \) evaluates to true in \( st \) and \( c \) preserves the truth value of \( b \), then \( \text{WHILE } b \text{ DO } c \text{ END} \) doesn’t terminate.

**Theorem while_no_terminate:**

\[
\forall (b: \text{bexp}) (c: \text{com}) (st: \text{state}),
\begin{align*}
&\forall st1 st2, st1 = [c] \Rightarrow st2 \Rightarrow \text{beval } st1 b = \text{beval } st2 b \Rightarrow \\
&\text{beval } st b = \text{true} \Rightarrow \\
&\neg (\exists st', st = [\text{WHILE } b \text{ DO } c \text{ END }] \Rightarrow st').
\end{align*}
\]

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

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

we must show the following proposition:

\[
\forall (cc : \text{com}) (st st' : \text{state}), \\
\text{st} = [cc] \Rightarrow st' \Rightarrow \\
cc = \text{WHILE } b \text{ DO } c \text{ END} \Rightarrow \\
\text{beval } st b = \text{true} \Rightarrow \\
\text{False}
\]

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.

---

\(^1\)Namely: turning \( \neg (\exists x, P) \) into \( \forall x, P \to \text{False} \), introducing \( cc \) to generalize the proposition about how the command evaluates, and shuffling some quantifiers around.
(The assumptions and goal are repeated verbatim on page 3 of the appendix for easy reference.)
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 \textit{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 \textit{all} possible subterms \( t \) of the appropriate type:

\[
\begin{align*}
t_1 &= (\forall x : \text{Bool}, \text{if } x \text{ then } \text{diverge} \text{ else } \text{false}) \ t \\
t_2 &= (\forall x : \text{Bool}, \text{if } (\text{if } x \text{ then } \text{false} \text{ else } \text{true}) \text{ then } \text{true} \text{ else } \text{diverge}) \ t
\end{align*}
\]

\[\square \quad \text{Coterminate} \quad \square \quad \text{Don’t coterminate}\]

\textit{Counterexample (if they don’t coterminate, give a value for }\ t)\textit{:}

\[
\begin{align*}
t_1 &= (\forall x : \text{Bool}, \text{if } (\text{if } x \text{ then } \text{true} \text{ else } \text{diverge}) \text{ then } \text{true} \text{ else } \text{diverge}) \ t \\
t_2 &= (\forall x : \text{Bool}, \text{diverge}) \ t
\end{align*}
\]

\[\square \quad \text{Coterminate} \quad \square \quad \text{Don’t coterminate}\]

\textit{Counterexample (if they don’t coterminate, give a value for }\ t)\textit{:}

\[
\begin{align*}
t_1 &= (\forall f : \text{Bool} \rightarrow \text{Bool}, f \text{ true}) \ t \\
t_2 &= (\forall f : \text{Bool} \rightarrow \text{Bool}, \text{true}) \ t
\end{align*}
\]

\[\square \quad \text{Coterminate} \quad \square \quad \text{Don’t coterminate}\]

\textit{Counterexample (if they don’t coterminate, give a value for }\ t)\textit{:}
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:

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

Under call by name, this steps to

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

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

By contrast, under call by value it steps to

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

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

Complete the `cbnstep` relation below so that it uses CBN reduction for function applications. (Note that `-n->` is notation for `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: *)
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.

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

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

(b) ☐ Terminates under CBN... ☐ Diverges under CBN
...and evaluates to:

$t = (\lambda f: \text{Bool} \to \text{Bool}, (\lambda y: \text{Bool}, f)) ((\lambda x: \text{Bool}, (\lambda z: \text{Bool}, x)) \text{ true})$

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

(b) ☐ Terminates under CBN... ☐ Diverges under CBN
...and evaluates to:

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

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

(b) ☐ Terminates under CBN... ☐ Diverges under CBN
...and evaluates to:
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:

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

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

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

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.)
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).

**Theorem cbn_progress :**

\[
\text{forall } t \in T,
\text{empty } |- t \in T \rightarrow
\text{value } t \lor (\text{exists } t', t \rightarrow t').
\]

**Proof:** By induction on \( t \).

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

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

**Case** \( t = \text{diverge} \)

**Case** \( t = \lambda x:T. t1 \)
Case $t = t_1 \ t_2$
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:

\[
\begin{align*}
\text{Theorem closed_under_1} : & \quad \forall (x \ y \ z : \text{string}), \\
& \quad \text{closed_under } [x;y;z] x.
\end{align*}
\]

\[
\begin{align*}
\text{Theorem closed_under_2} : & \quad \forall (x \ y \ z : \text{string}), \\
& \quad \text{closed_under } [] \langle \langle x : \text{Bool}, x \rangle \rangle.
\end{align*}
\]

\[
\begin{align*}
\text{Theorem closed_under_3} : & \quad \forall (x : \text{string}), \\
& \quad \neg \text{closed_under } [] x.
\end{align*}
\]

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

\[
\begin{align*}
\text{Inductive closed_under} : \text{list string } \to \text{tm } \to \text{Prop} :=
\end{align*}
\]
Write a careful informal proof of the following theorem, which relates \texttt{appears\_free\_in} to the new definition of \texttt{closed\_under\_env} from the previous question. Intuitively, the theorem states that, if all variables that appear free in \( t \) are in the list \( \text{env} \), then \( t \) is closed under \( \text{env} \).

**Theorem** \( \text{appears\_free\_in\_closed\_under} : \)

\[
\forall t \text{ env}, \quad (\forall x, \text{appears\_free\_in} x t \rightarrow \text{In} x \text{ env}) \rightarrow \text{closed\_under} \text{ env} t.
\]

Again, give just the cases for \textit{variables}, \textit{applications}, and \textit{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:**
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 \( \text{Top} \). On the other hand, the type \( (\text{Top} \times \text{Top}) \times (\text{Top} \times \text{Top}) \) has exactly four proper supertypes:

\[
\begin{align*}
(\text{Top} \times \text{Top}) & \times \text{Top} \\
\text{Top} & \times (\text{Top} \times \text{Top}) \\
\text{Top} & \times \text{Top} \\
\text{Top} & \times \text{Top}
\end{align*}
\]

11.1 Using just the \( \text{Top} \) 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></td>
</tr>
<tr>
<td>1</td>
<td>( \text{Top} \times \text{Top} )</td>
</tr>
<tr>
<td>2</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>( (\text{Top} \times \text{Top}) \times (\text{Top} \times \text{Top}) )</td>
</tr>
</tbody>
</table>

11.2 Using just the \( \text{Top} \) and \( \text{Bool} \) types and the \( \text{arrow} \) 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></td>
</tr>
<tr>
<td>1</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></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).

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).
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 Top, which is a supertype of all other types, as indicated by this subtyping rule:

\[ S <: Top \]

\[ S <: \text{Top} \]  

(S_Top)

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

\[ Bot <: S \]

\[ Bot <: S \]  

(S_Bot)

Just as when we added 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 Bot as described above:

- **Lemma sub_inversion_Bot**: \( \forall U, U <: Bot \rightarrow U = Bot \).
  - Provable
  - Not provable

- **Lemma sub_inversion_Bool**: \( \forall U, U <: Bool \rightarrow U = Bool \).
  - Provable
  - Not provable

- **Lemma canonical_forms_of_Bool**: \( \forall s, \emptyset \vdash s \in Bool \rightarrow \text{value } s \rightarrow (s = ttrue \lor s = tfalse) \).
  - Provable
  - Not provable

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

\[ \emptyset \vdash v : Bot \]

If so, give an example. If not, briefly explain.
(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 \(<:\)).

- \( T = \text{Bool} \rightarrow \text{Top} \quad S = \text{Bot} \rightarrow \text{Bot} \)
  - \( \square T <: S \)
  - \( \square S <: T \)
  - \( \square \text{incomparable} \)

- \( T = (\text{Bot} \rightarrow \text{Top}) \rightarrow \text{Bool} \quad S = (\text{Top} \rightarrow \text{Bot}) \rightarrow \text{Top} \)
  - \( \square T <: S \)
  - \( \square S <: T \)
  - \( \square \text{incomparable} \)

- \( T = \text{Bool} \rightarrow (\text{Top} \ast \text{Bot}) \quad S = \text{Bot} \rightarrow (\text{Bot} \ast \text{Top}) \)
  - \( \square T <: S \)
  - \( \square S <: T \)
  - \( \square \text{incomparable} \)

(d) Consider the following program:

\[
\text{empty} \vdash (\lambda x : T. x \ x) : T \rightarrow \text{Bot}
\]

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?

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

Theorem Sn_le_Sm__n_le_m : forall n m,
S n <= S m -> n <= m.

Minimum and Maximum

Definition min (a : nat) (b : nat) :=
if a <=? b then a else b.

Definition max (a : nat) (b : nat) :=
if a <=? b then b else a.

Imp

Syntax:

Inductive aexp : Type :=
| ANum (n : nat)
| APlus (a1 a2 : aexp)
| AMinus (a1 a2 : aexp)
| AMult (a1 a2 : aexp).

Inductive bexp : Type :=
| BTrue
| BFalse
| BEq (a1 a2 : aexp)
| BLe (a1 a2 : aexp)
| BNot (b : bexp)
| BAnd (b1 b2 : bexp).

Inductive com : Type :=
| CSkip
| CAss (x : string) (a : aexp)
| CSeq (c1 c2 : com)
| CIf (b : bexp) (c1 c2 : com)
| CWhile (b : bexp) (c : 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:

\[
\text{Inductive ceval : com -> state -> state -> Prop :=}
\]

| E_Skip : \forall st, st = [\text{SKIP}] \Rightarrow st |
| E_Ass : \forall st a1 n x, aeval st a1 = n \Rightarrow st = [x ::= a1] \Rightarrow (x !-> n ; st) |
| E_Seq : \forall c1 c2 st st', st = [c1] \Rightarrow st' \Rightarrow st = [c1 ;; c2] \Rightarrow st' |
| E_IfTrue : \forall st st' b c1 c2, beval st b = true \Rightarrow st = [c1] \Rightarrow st' \Rightarrow st = [\text{TEST b THEN c1 ELSE c2 FI}] \Rightarrow st' |
| E_IfFalse : \forall st st' b c1 c2, beval st b = false \Rightarrow st = [c2] \Rightarrow st' \Rightarrow st = [\text{TEST b THEN c1 ELSE c2 FI}] \Rightarrow st' |
| E_WhileFalse : \forall b st c, beval st b = false \Rightarrow st = [\text{WHILE b DO c END}] \Rightarrow st |
| E_WhileTrue : \forall st st' st' b c, beval st b = true \Rightarrow st = [c] \Rightarrow st' \Rightarrow st' = [\text{WHILE b DO c END}] \Rightarrow st'' \Rightarrow st = [\text{WHILE b DO c END}] \Rightarrow st'' |

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

Problem 6: Assumptions and Goal

Assumptions:

- b : bexp
- c : com
- Hpres : \forall st1 st2, st1 = [c] \Rightarrow st2 \Rightarrow beval st1 b = beval st2 b

Goal:

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

Syntax

\[ T ::= \text{Bool} \]
\[ t ::= x \]
\[ v ::= \text{true} \]
\[ | T \rightarrow T \]
\[ | t \rightarrow t \]
\[ | \lambda x : T, t \rightarrow \lambda x : T, t \]
\[ | \text{true} \rightarrow \text{true} \]
\[ | \text{false} \rightarrow \text{false} \]
\[ | \text{if } t \text{ then } t \text{ else } t \rightarrow \text{if } t \text{ then } t \text{ else } t \]

Small-step operational semantics

\[
\begin{array}{l}
\text{value } v \\
\hline
\text{(ST_AppAbs)} \quad \text{(\(\lambda x : T, t\)) } v \rightarrow [x := v]t
\end{array}
\]

\[
\begin{array}{l}
t1 \rightarrow t'1 \\
\hline
\text{(ST_App1)} \\
\text{t1 t2 \rightarrow t'1' t2} \\
\hline
\text{(ST_App2)} \\
\text{value v1 t2 \rightarrow v1 t2'}
\end{array}
\]

\[
\begin{array}{l}
\text{(if true then t1 else t2) \rightarrow t1} \\
\hline
\text{(ST_IfTrue)}
\end{array}
\]

\[
\begin{array}{l}
\text{(if false then t1 else t2) \rightarrow t2} \\
\hline
\text{(ST_IfFalse)}
\end{array}
\]

\[
\begin{array}{l}
t1 \rightarrow t'1 \\
\hline
\text{(ST_If)} \\
\text{(if t1 then t2 else t3) \rightarrow (if t1' then t2 else t3)}
\end{array}
\]

Typing

\[
\begin{array}{l}
\text{Gamma } x = T \\
\hline
\text{(T_Var)} \\
\text{Gamma, x : T11 |- t12 } \in T12
\end{array}
\]

\[
\begin{array}{l}
\text{Gamma } |- x \in T \\
\hline
\text{(T_Var)} \\
\text{Gamma, x : T11, t12 } \in T11 \rightarrow T12
\end{array}
\]

\[
\begin{array}{l}
\text{Gamma } |- t1 \in T11 \rightarrow T12 \\
\hline
\text{(T_App)} \\
\text{Gamma } |- t2 \in T11
\end{array}
\]

\[
\begin{array}{l}
\text{Gamma } |- t1 \in T11 \rightarrow T12 \\
\hline
\text{(T_App)} \\
\text{Gamma } |- t2 \in T12
\end{array}
\]

\[
\begin{array}{l}
\text{Gamma } |- t1 \text{ \in T12} \\
\hline
\text{(T_App)} \\
\text{Gamma } |- \text{true } \in \text{Bool}
\end{array}
\]

\[
\begin{array}{l}
\text{Gamma } |- \text{false } \in \text{Bool}
\end{array}
\]

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

\[
\text{Inductive appears\_free\_in : string -> tm -> Prop :=}
\]
\[
\begin{align*}
| \text{afi\_var : forall x,} & \text{ appears\_free\_in x} \text{ } \langle\langle x\rangle\rangle \\
| \text{afi\_app1 : forall x t1 t2,} & \text{ appears\_free\_in x} \text{ } t1 \text{ } \rightarrow \\
& \text{ appears\_free\_in x} \text{ } \langle\langle t1 t2\rangle\rangle \\
| \text{afi\_app2 : forall x t1 t2,} & \text{ appears\_free\_in x} \text{ } t2 \text{ } \rightarrow \\
& \text{ appears\_free\_in x} \text{ } \langle\langle t1 t2\rangle\rangle \\
| \text{afi\_abs : forall x y T11 t12,} & \text{ y} \text{ } \neq \text{ } x \text{ } \rightarrow \\
& \text{ appears\_free\_in x} \text{ } t12 \text{ } \rightarrow \\
& \text{ appears\_free\_in x} \text{ } \langle\langle y:T11, t12\rangle\rangle \\
| \text{afi\_if1 : forall x t1 t2 t3,} & \text{ appears\_free\_in x} \text{ } t1 \text{ } \rightarrow \\
& \text{ appears\_free\_in x} \text{ } t2 \text{ } \rightarrow \\
& \text{ appears\_free\_in x} \text{ } \langle\langle \text{if t1 then t2 else t3}\rangle\rangle \\
| \text{afi\_if2 : forall x t1 t2 t3,} & \text{ appears\_free\_in x} \text{ } t1 \text{ } \rightarrow \\
& \text{ appears\_free\_in x} \text{ } t2 \text{ } \rightarrow \\
& \text{ appears\_free\_in x} \text{ } \langle\langle \text{if t1 then t2 else t3}\rangle\rangle \\
| \text{afi\_if3 : forall x t1 t2 t3,} & \text{ appears\_free\_in x} \text{ } t3 \text{ } \rightarrow \\
& \text{ appears\_free\_in x} \text{ } \langle\langle \text{if t1 then t2 else t3}\rangle\rangle .
\end{align*}
\]

\text{Definition old\_closed (t:tm) :=}
\[
\text{forall x, \sim appears\_free\_in x} \text{ } t .
\]

\text{Hint Constructors appears\_free\_in.}

Values for STLC with Bools

\[
\text{Inductive value : tm -> Prop :=}
\]
\[
\begin{align*}
| \text{v\_abs : forall x T t,} & \text{ value} \text{ } \langle\langle x:T, t\rangle\rangle \\
| \text{v\_true :} & \text{ value} \text{ } \langle\langle \text{true}\rangle\rangle \\
| \text{v\_false :} & \text{ value} \text{ } \langle\langle \text{false}\rangle\rangle .
\end{align*}
\]

Properties of STLC

\[
\text{Theorem preservation : forall t t' T,}
\]
\[
\text{empty} \text{ } |- \text{ } t \text{ } \in T \text{ } \rightarrow \\
\text{empty} \text{ } |- \text{ } t' \text{ } \in T .
\]

(Continued on next page.)
Theorem progress : forall t T,
empty |- t \in T ->
value t \ / exists t', t --> t'.
STLC with Booleans, Products and Subtyping

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

\[
T ::= \ldots \\
    \mathbin{\top} \\
\]

Add these rules that characterize the subtyping relation:

\[
\begin{align*}
    S <: U \quad U <: T & \quad \text{(S_Trans)} \quad T <: T \quad S <: \mathbin{\top} \quad \text{(S_Top)} \\
    S1 <: T1 \quad S2 <: T2 & \quad \text{(S_Prod)} \\
    S1 \rightarrow S2 <: T1 \rightarrow T2 & \quad \text{(S_Arrow)}
\end{align*}
\]

And add this to the typing relation:

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