1. **Inductive relations** (11 points)

Complete the definition at the bottom of the page of an Inductive relation `count` that relates a list to the number of elements of a list satisfying some predicate `P` — that is, `count P l n` should be true when the number of elements of `l` satisfying `P` is `n`. For example, if we define

\[
\text{Definition iszero} \ (n : \text{nat}) : \text{Prop} := (n = 0).
\]

then the propositions

- `count iszero [] 0`
- `count iszero [1,2,3] 0`
- `count iszero [0,1,2,3] 1`
- `count iszero [1,0,0,2,3,0] 3`

should all be provable, whereas the propositions

- `count iszero [1,2,3] 3`
- `count iszero [0,0] 4`
- `count iszero [] 1`

should not be provable.

**Answer:**

\[
\text{Inductive count} \ (X : \text{Type}) \ (P : X \rightarrow \text{Prop}) : \text{list} \ X \rightarrow \text{nat} \rightarrow \text{Prop} :=
\]

\[
\begin{align*}
R0 &: \text{count P nil 0} \\
| Rno &: \forall l n x, \text{count P l n} \rightarrow \neg(P \ x) \rightarrow \text{count P} \ (x::l) \ n \\
| Ryes &: \forall l n x, \text{count P l n} \rightarrow P \ x \rightarrow \text{count P} \ (x::l) \ (S \ n).
\end{align*}
\]

2. **Termination in Coq** (12 points)

We know that every Fixpoint definition in Coq must pass a “termination checker,” to ensure that all definable functions terminate on all inputs.

**2.1** For example, here is the definition of the function `app` which concatenates (appends) two lists:

\[
\text{Fixpoint app} \ (l1 \ l2 : \text{natlist}) : \text{natlist} :=
\]

\[
\begin{align*}
&\quad \text{match} \ l1 \ \text{with} \\
&\quad | \ \text{nil} \Rightarrow \ l2 \\
&\quad | \ h :: t \Rightarrow h :: (\text{app} \ t \ l2)
\end{align*}
\]
In one or two sentences, explain how Coq is able to automatically deduce that this function always terminates.

Answer: Coq’s termination checker looks for an argument to the recursive function that is “obviously decreasing” on every recursive call—that is, the value at the recursive call site is extracted by pattern matching from the value given as a parameter to the function.

Here, whenever \texttt{app} calls itself recursively, the first argument \texttt{l1} has the shape \texttt{h :: t}. The first argument in the recursive call (\texttt{t}) is strictly smaller than \texttt{l1}.

2.2 Fill in a Fixpoint definition (it doesn’t have to make sense) that would not pass Coq’s termination checker.

\texttt{Fixpoint nonsense (l1 l2 : natlist) : natlist :=}

\texttt{nonsense l2 l1.}

2.3 Although functions defined with \texttt{Fixpoint} must terminate in Coq, this isn’t true in Coq’s tactic language \texttt{Ltac}. Indeed, it is possible to write tactics that will loop forever.

Should we be worried about this, from the perspective of logical consistency? Briefly explain why or why not.

We should not be worried. Coq’s logic would be inconsistent if we could construct a nonterminating Gallina expressions of arbitrary types (and this would be worrying!). But when an tactic script fails to terminate, no such Gallina expression gets built—we simply fail to prove whatever theorem was the goal at this point.

3. Program equivalence (12 points)

Consider the Imp language from chapter \texttt{Imp.v} of Software Foundations (summarized on pages ?? and ?? in the appendix). The definition of equivalence for Imp programs can be rephrased as follows:

- Programs \texttt{c1} and \texttt{c2} are called \textit{equivalent on state} \texttt{st} if, for every state \texttt{st’}, \texttt{c1 / st \ \st’} if and only if \texttt{c2 / st \ \st’}.
- Programs \texttt{c1} and \texttt{c2} are \textit{equivalent} if they are equivalent on every state.
- Programs \texttt{c1} and \texttt{c2} are called \textit{somewhat inequivalent} if there is some state \texttt{st1} on which they are equivalent and another state \texttt{st2} on which they are not equivalent.
- Programs \texttt{c1} and \texttt{c2} are called \textit{very inequivalent} if there is no state \texttt{st} on which they are equivalent.

For each of the following pairs of programs \texttt{c1} and \texttt{c2}, check the appropriate box to indicate whether they are \textit{equivalent}, \textit{somewhat inequivalent}, or \textit{very inequivalent}. (Keep in mind that Imp variables hold natural numbers (\(\geq 0\)), not integers.) If you choose \textit{somewhat inequivalent}, give an example of a starting state \texttt{st} on which \texttt{c1} and \texttt{c2} behave differently.
3.1 \[ \begin{align*} &c_1 = \text{WHILE False DO } X ::= X + 1 \text{ END} \\ &c_2 = \text{WHILE } X > 0 \text{ DO } X ::= X + 1 \text{ END} \end{align*} \]

\[ \begin{array}{ccc} \square \text{ Equivalent} & \times \text{ Somewhat inequivalent} & \square \text{ Very inequivalent} \\ \end{array} \]

For example: \{X \rightarrow 1\}

3.2 \[ \begin{align*} &c_1 = \text{WHILE } X > 0 \text{ DO } X ::= X + 1 \text{ END} \\ &c_2 = \text{WHILE } X > 0 \text{ DO } Y ::= Y + 1 \text{ END} \end{align*} \]

\[ \begin{array}{ccc} \times \text{ Equivalent} & \square \text{ Somewhat inequivalent} & \square \text{ Very inequivalent} \\ \end{array} \]

For example:

3.3 \[ \begin{align*} &c_1 = X ::= X + 1 \text{ ;; } Y ::= X \\ &c_2 = Y ::= X + 1 \end{align*} \]

\[ \begin{array}{ccc} \square \text{ Equivalent} & \square \text{ Somewhat inequivalent} & \times \text{ Very inequivalent} \\ \end{array} \]

For example:

3.4 \[ \begin{align*} &c_1 = \text{IFB } X = Y \text{ THEN } X ::= X - Y \text{ ELSE } X ::= 0 \text{ FI} \\ &c_2 = \text{SKIP} \end{align*} \]

\[ \begin{array}{ccc} \square \text{ Equivalent} & \times \text{ Somewhat inequivalent} & \square \text{ Very inequivalent} \\ \end{array} \]

For example: \{X \rightarrow 2; Y \rightarrow 3\}

3.5 \[ \begin{align*} &c_1 = \text{IFB } X = Y \text{ THEN } X ::= X - Y \text{ ELSE } X ::= 0 \text{ FI} \\ &c_2 = \text{WHILE } X > 0 \text{ DO } X ::= X - 1 \text{ END} \end{align*} \]

\[ \begin{array}{ccc} \times \text{ Equivalent} & \square \text{ Somewhat inequivalent} & \square \text{ Very inequivalent} \\ \end{array} \]

For example:

4. [Standard Only] Hoare triples (12 points)

Each of the following Hoare triples contain a variable \(c\), representing an arbitrary Imp program (that is, a com). For each triple, check the appropriate box to indicate whether the triple is always valid (valid for all choices of \(c\)), sometimes valid (valid for some, but not all, choices of \(c\)), or never valid (invalid for all choices of \(c\)). If you choose sometimes valid, give an example of a command \(c_1\) that makes the triple valid and a command \(c_2\) that makes the triple invalid.

4.1 \[
\begin{align*}
\{X = 3\} &\\
\text{WHILE True DO } c \text{ END} &\\
\{X = 3\} &\\
\end{align*}
\]

\[ \begin{array}{ccc} \times \text{ Always valid} & \square \text{ Sometimes valid} & \square \text{ Never valid} \\ \end{array} \]

\[c_1 = \]
\[c_2 =\]

4.2 \[
\begin{align*}
\{X = 1\} &\\
\text{WHILE } X > 0 \text{ DO } c \text{ END } ;; &\\
\text{WHILE } X = 0 \text{ DO } c \text{ END} &\\
\end{align*}
\]

3
{{ False }}

□ Always valid ☒ Sometimes valid □ Never valid

c1 = SKIP

c2 = IFB X = 1 THEN X ::= 0 ELSE X ::= 1

4.3

{{ True }}

WHILE X > 0 DO c ;; Y ::= 1 END ;;
WHILE X = 0 DO c ;; Y ::= 0 END

{{ Y = 0 }}

☒ Always valid □ Sometimes valid □ Never valid

c1 =
c2 =

4.4

{{ Y = X }}

IFB X = 0 THEN
  SKIP
ELSE
  WHILE X > 0 DO c END

{{ Y > 0 }}

□ Always valid □ Sometimes valid ☒ Never valid

c1 =
c2 =

4.5

{{ X = Y }}

;c ;c

{{ X = Y + 1 }}

□ Always valid ☒ Sometimes valid □ Never valid

c1 = X ::= 1 ;; Y ::= 0
c2 = SKIP

5. Loop invariants (12 points)

For each pair of Hoare triple and proposed loop invariant Inv, your job is to decide whether Inv can be used to prove a Hoare triple of this form:

{{P}} WHILE b DO c END {{Q}}

Specifically, you should decide whether Inv satisfies each of the specific constraints

Implied by precondition: P \rightarrow Inv

Preserved by loop body: \{ Inv \land b \} c \{ Inv \}

Implies postcondition: (Inv \land \neg b) \rightarrow Q

from the Hoare rule for WHILE.
5.1 \{\{ X = 3 \}\} Inv: X \neq 4

\begin{verbatim}
WHILE X * X < 10 DO
    X ::= X + 2
END
\{\{ X = 5 \}\}
\end{verbatim}

(a) Implied by precondition
\begin{itemize}
    \item True
    \item False
\end{itemize}

(b) Preserved by loop body
\begin{itemize}
    \item False
    \item False
\end{itemize}

(c) Implies postcondition
\begin{itemize}
    \item True
    \item False
\end{itemize}

5.2 \{\{ X = 0 \}\} Inv: X = 0

\begin{verbatim}
WHILE X < Y DO
    X ::= Y - X ;
    Y ::= Y - X
END
\{\{ Y = 0 \}\}
\end{verbatim}

(a) Implied by precondition
\begin{itemize}
    \item True
    \item False
\end{itemize}

(b) Preserved by loop body
\begin{itemize}
    \item False
    \item False
\end{itemize}

(c) Implies postcondition
\begin{itemize}
    \item True
    \item False
\end{itemize}

5.3 \{\{ X = 0 \}\} Inv: True

\begin{verbatim}
WHILE X = 0 DO
    SKIP
END
\{\{ X \neq 0 \}\}
\end{verbatim}

(a) Implied by precondition
\begin{itemize}
    \item True
    \item False
\end{itemize}

(b) Preserved by loop body
\begin{itemize}
    \item True
    \item False
\end{itemize}

(c) Implies postcondition
\begin{itemize}
    \item True
    \item False
\end{itemize}
5.4 \[\{\{ X > 0 \}\}\] \hspace{1cm} \text{Inv: False}

\[\text{WHILE } X > 0 \text{ DO}\]
\[X ::= X + 1\]
\[\text{END}\]
\[\{\{ X > 0 \}\}\]

(a) Implied by precondition
\[\square \text{ True} \hspace{1cm} \otimes \text{ False}\]

(b) Preserved by loop body
\[\otimes \text{ True} \hspace{1cm} \square \text{ False}\]

(c) Implies postcondition
\[\otimes \text{ True} \hspace{1cm} \square \text{ False}\]

6. \textbf{[Advanced Only] Hoare logic} (16 points)

In \texttt{Hoare.v}, we experimented with extending Imp with a \texttt{REPEAT} command, written either \texttt{CRepeat b c} or \texttt{REPEAT c UNTIL b END}. The meaning of this command is to execute the command \texttt{c} \textit{one or more} times, until the condition \texttt{b} is satisfied. That is, \textit{after} each time \texttt{c} is executed, we check if \texttt{b} holds; if it does hold, we halt the loop.

The \texttt{ceval} relation (repeated on page 2 of the appendix) is extended with the following rules:

\[\text{E RepeatEnd : forall st st' b1 c1, ceval st c1 st' -> beval st' b1 = true -> ceval st (CRepeat c1 b1) st'}\]
\[\text{E RepeatLoop : forall st st' st'' b1 c1, ceval st c1 st' -> beval st' b1 = false -> ceval st' (CRepeat c1 b1) st'' -> ceval st (CRepeat c1 b1) st''}\]

Here is one possible Hoare-logic rule for \texttt{REPEAT} loops:

\[
\frac{\{ P \} \ c \ \{ P \}}{\{ P \} \ \text{REPEAT} \ c \ \text{UNTIL} \ b \ \text{END} \ \{ P \ \land \ b \}} \tag{hoare_repeat}
\]

Write a careful informal proof of the correctness of this rule.

Please use full, grammatical sentences. If you use induction, be explicit about the induction hypothesis. If you run out of space use the following blank page and write “continued” at the bottom of this page so we know to look for the rest (Gradescope only displays one page at a time).

\textbf{Proof:} Suppose \texttt{st} is a state satisfying \texttt{P} and that \texttt{(REPEAT c UNTIL b END) / st \downarrow st'}. We must then show \texttt{(P \land b) / st'}. Proceed by induction on a derivation of \texttt{(REPEAT c UNTIL b END) / st \downarrow st'}. By the form of the program, there are just two possibilities:
(a) Suppose \((\text{REPEAT } c \text{ UNTIL } b \text{ END}) / st \downarrow st'\) by rule \(E_{\text{RepeatEnd}}\), with \(c / st \downarrow st'\) and beval \(st' b = \text{true}\). Since \(P st\), the assumption \(\{P\} c \{P\}\) gives us \(P st'\), and the assertion \(b st'\) follows from beval \(st' b = \text{true}\), so \(st'\) satisfies the required postcondition.

(b) Suppose \((\text{REPEAT } c \text{ UNTIL } b \text{ END}) / st \downarrow st'\) by rule \(E_{\text{RepeatLoop}}\), with \(c / st \downarrow st1\) and beval \(st1 b = \text{false}\) and \((\text{REPEAT } c \text{ UNTIL } b \text{ END}) / st1 \downarrow st'\). By the induction hypothesis, \((P \land b) st'\). Since \(P st\), the assumption \(\{P\} c \{P\}\) again gives us \(P st1\), and the induction hypothesis immediately yields boty \(P st'\) and \(b st'\), as required.

For comparison, here is the proof in Coq:

Theorem hoare_repeat_easyversion : forall P b c,  
    \{P\} c \{P\} ->  
    \{P\} \text{REPEAT } c \text{ UNTIL } b \text{ END} \{\text{fun } st \Rightarrow P st \land (\text{bassn b st})\}.
Proof.  
  intros P b c Hhoare st st' He HP.  
  remember (\text{REPEAT } c \text{ UNTIL } b \text{ END}) as wcom eqn:Hwcom.  
  induction He; inversion Hwcom; subst; clear Hwcom.  
  split.  
  + apply (Hhoare st); assumption.  
  + apply H.  
  - apply IHHe2; try reflexivity. apply (Hhoare st); assumption.  
Qed.

Note that no generalization on \(st\) or \(st'\) is needed. In the second bullet (the one needing the induction hypothesis), two induction hypotheses are generated (one of them nonsensical)

\[\text{IHHe}2 : (\text{REPEAT } c \text{ UNTIL } b \text{ END}) = (\text{REPEAT } c \text{ UNTIL } b \text{ END}) -> P st' -> P st'\land (\text{bassn b st'})\]
\[\text{IHHe1 : } c = (\text{REPEAT } c \text{ UNTIL } b \text{ END}) -> P st -> P st'\land (\text{bassn b st'})\]

and the goal in this case is:

\[P st'' \land (\text{bassn b st''})\]

We can see here that the generated \(\text{IHHe}2\) gives us exactly what we want.

7. **[Standard Only] Small-step reduction** (12 points)

Consider the simply typed lambda-calculus with booleans and conditionals, defined on pages ?? and ??.

Here are three examples of STLC terms. The following questions ask you to decide which of these terms are values, and then to answer how each of the three terms steps.

<table>
<thead>
<tr>
<th>Coq notation:</th>
<th>Informal notation:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Example (e1 := \text{tabs } x \text{TBool } (\text{tapp } (\text{tvar } x) (\text{tvar } x))) (\backslash x: T. x x)</td>
<td>(e1 \text{ e1 e1 e1})</td>
</tr>
<tr>
<td>Example (e2 := \text{tapp } e1 e1)</td>
<td>if (e1) \text{ true then } (e2) \text{ else } (e2)</td>
</tr>
<tr>
<td>Example (e3 := \text{tif } (\text{tapp } e1 \text{ ttrue}) e2 e2)</td>
<td></td>
</tr>
</tbody>
</table>

(Those of these terms are actually well-typed, but this does not prevent us from reasoning about their reduction behavior, which is all we are interested in here.)
7.1 Which of the above terms are values?

- e1  □ e2  □ e3

7.2 List all the terms that e1 multi-steps to.

*Answer*: e1

7.3 What term (if any) does e2 single-step to?

*Answer*: e2

7.4 List all the terms that e3 multi-steps to.

*Answer*: Two terms: if (e1 true) then e2 else e2 and if (true true) then e2 else e2.

8. [Advanced Only] Big-step and small-step semantics (16 points)

Again, consider the simply typed lambda-calculus with booleans and conditionals (pages ?? and ?? in the appendix). An alternative big-step evaluation relation for the same language is given on page ??

Give careful informal proofs of the following theorem and the easy corollary below. The beginning of each proof is given; please fill in the requested cases.

**Theorem 1**: If t ==> s and s \ v, then t \ v.

**Proof**: By induction on a derivation of t ==> s.

Give just the cases where the last rule in the derivation is ST_AppAbs or ST_App1. (The other cases are similar.)

- If the last rule in the derivation is ST_AppAbs, then t has the form \[x:T.t12 v2\], where v2 is a value, and s = [x:=v2]t12 By E_Value, \[x:T.t12 v2 \ \ \ \ \ x:T.t12 v2 and v2 \ \ v2\]. But then E_App (using the assumption that s \ v, where s = [x:=v2]t12) yields t \ s.

- If the last rule is ST_App1, then t has the form t1 t2, and t1 ==> t1'. From the assumption that t1' t2 \ v and the fact that E_App is the only rule that can be used to prove this conclusion, we obtain t1' \ x:T.t12 and t2 \ v2, and [x:=v2]t12 \ v. By the induction hypothesis, t1 \ x:T.t12. But then by E_App, t1 t2 \ v, as required.

**Corollary 2**: If t ==>* v and v is a value, then t \ v.

**Proof**: By induction on a derivation of t ==>* v— that is, a derivation of multi step t v, where step is the single-step reduction relation from page ?? and multi was defined as follows:

```
Inductive multi {X:Type} (R: relation X) : relation X :=
| multi_refl : forall (x : X), multi R x x
| multi_step : forall (x y z : X),
```
\[ R \times y \rightarrow \\
\text{multi } R \ y \ z \rightarrow \\
\text{multi } R \ x \ z. \]

- Suppose the last step in the derivation is rule \textbf{multi_refl}. Then \( t = v \) and rule \textbf{E_Value} immediately yields \( t \ \\bot \ \bot v \).
- Suppose the last step in the derivation is rule \textbf{multi_step}, with \( t \Rightarrow t' \) and \( t' \Rightarrow* v \).
  By the induction hypothesis, \( t' \ \\bot \ \bot v \). By Theorem 1, \( t \ \\bot \ \bot v \).

9. \textbf{Properties of reduction and typing} (16 points)

Suppose we are given some new programming language — i.e., someone specifies

- a set of terms \textbf{Inductive} \( \text{tm} \),
- a property \( \text{value} : \ \text{tm} \rightarrow \text{Prop} \) that picks out a subset of terms designated as values,
- a small-step reduction relation \textbf{Inductive} \( \text{stepsto} : \ \text{tm} \rightarrow \text{tm} \rightarrow \text{Prop} \),
- a set of types \textbf{Inductive} \( \text{ty} \), and
- a typing relation \textbf{Inductive} \( \text{hastype} : \ \text{tm} \rightarrow \text{ty} \rightarrow \text{Prop} \).

For simplicity, let's suppose that this language doesn't have any variable binders, so we don't need any contexts, and typing is just a two-place relation. Please also assume that the typing relation makes no mention of the step relation, and vice versa.

We use lower-case variables like \( t \) to stand for terms (in \( \text{tm} \)) and upper-case variables like \( T \) to stand for types (in \( \text{ty} \)). We write \( t1 \Rightarrow t2 \) to mean \( \text{stepsto } t1 \ t2 \) and \( \vdash t \in T \) to mean \( \text{hastype } t \ T \).

Further, suppose that we are told the following facts about this language:

- \textbf{Uniqueness of typing}: If \( t \) is a term and \( T1, T2 \) are types such that \( \vdash t \in T1 \) and \( \vdash t \in T2 \), then \( T1 = T2 \).
- \textbf{Determinism of reduction}: If \( t \Rightarrow t1 \) and \( t \Rightarrow t2 \), then \( t1 = t2 \).
- \textbf{Progress}: If \( \vdash t \in T \), then either \( t \) is a value or there is some \( t' \) such that \( t \Rightarrow t' \).
- \textbf{Preservation}: If \( \vdash t \in T \) and \( t \Rightarrow t' \), then \( \vdash t' \in T \).
- \textbf{Values are normal forms}: If \( t \) is a value, then there is no \( t' \) such that \( t \Rightarrow t' \).

In each of the following parts, we ask you to consider how a proposed change to this language will affect these properties (without knowing anything more about the details of the language). If the proposed change will \textit{definitely break} the corresponding property, check the box by “fails.” If the proposed change definitely \textit{cannot break} the property, check the box next to “holds.” If this change \textit{might or might not break} the property, depending on the details of the original language and/or exactly what is added or removed, choose “depends.”
9.1 If we restrict the typing relation (i.e., we take one or more pairs of a term \( t \) and a type \( T \) with \( |- t \\in T \) and remove them from the relation), what happens to these properties?

Uniqueness of typing: ☒ holds ☐ fails ☐ depends
Determinism of reduction: ☒ holds ☐ fails ☐ depends
Progress: ☐ holds ☐ fails ☒ depends
Preservation: ☐ holds ☐ fails ☒ depends

Typing only gets “more unique.” Reduction is not affected. Progress is easy because it mentions the typing relation only in a hypothesis, so restricting typing obviously makes the relation “more true.” Preservation, though, has typing on both sides of an arrow. So, for example, if we remove the pair \( |- \text{true} \\in \text{Bool} \) from the typing relation, then the term 
if true then true else true will still be well typed, but it will reduce to true, which is not. Conversely, if we remove all pairs from the typing relation (leaving it empty), then preservation still holds (trivially).

9.2 If we enlarge the typing relation (i.e., we add one or more new pairs \( |- t \in T \) that were not in the original relation), what happens?

Uniqueness of typing: ☐ holds ☐ fails ☒ depends
Determinism of reduction: ☒ holds ☐ fails ☐ depends
Progress: ☒ holds ☐ fails ☐ depends
Preservation: ☒ holds ☐ fails ☒ depends

Uniqueness of typing might either fail (if we add a new pair \( |- t \in T \) where \( t \) was already well typed (but with a different type \( T \)) or continue to hold (if we only add pairs \( |- t \in T \) where \( t \) was not typeable before). Determinism of reduction is unaffected by changes to typing. Progress may fail (if a non-value normal form is given a type) or not. Preservation may fail (if we give a type to a non-typeable term that also reduces to a non-typeable term) or not.

9.3 If we restrict the reduction relation for well-typed terms (i.e., we take one or more pairs of terms \( t \) and \( t' \) with \( t \Rightarrow t' \) and \( |- t \in T \) for some type \( T \) and remove them from the reduction relation), what happens?

Uniqueness of typing: ☒ holds ☐ fails ☐ depends
Determinism of reduction: ☒ holds ☐ fails ☐ depends
Progress: ☐ holds ☒ fails ☐ depends
Preservation: ☐ holds ☒ fails ☐ depends
Changing reduction doesn’t affect uniqueness of typing. Reduction will continue to be deterministic if we only restrict it. Progress must fail because we’ve assumed that values are already normal forms. Preservation continues to hold because this change only makes its premise harder to satisfy.

9.4 If we enlarge the reduction relation for well-typed terms (i.e., we add one or more new pairs \( t \mapsto t' \), where \( |- t \in T \) is in the original typing relation for some type \( T \)), what happens?

Uniqueness of typing: ☒ holds ☐ fails ☐ depends

Determinism of reduction: ☐ holds ☐ fails ☒ depends

Progress: ☒ holds ☐ fails ☐ depends

Preservation: ☐ holds ☐ fails ☒ depends

Changing reduction doesn’t affect uniqueness of typing. Reduction may either continue to be deterministic (if we only add reduction steps for terms that were previously values) or fail. Progress will continue to hold because this change only makes its right-hand side easier to satisfy. Preservation may either continue to hold (if we only add pairs \( t \mapsto t' \) where \( t \) and \( t' \) both have the same type or are both not typable).

10. Simply Typed Lambda Calculus (10 points)

10.1 Is it possible to find a term \( t \) and a type \( T \) such that \( |- t \in T \) in the ordinary simply typed lambda-calculus with booleans and pairs (without subtyping) (pages ?? to 9 in the appendix)? If so, give an example. If not, explain why this cannot occur.

☐ Yes ☒ No

(To see why this is not possible, suppose \( t \) were such a term and \( T \) its type. For the application \( t \, t \) to be well typed, \( t \) must have an arrow type, and its left-hand side must match the (unique) type of \( t \)—that is, we must have \( T = T \rightarrow U \) for some \( U \). Such a \( T \) cannot not exist—it would have to be infinitely long.)

10.2 Is it possible to find a term \( t \) and a type \( T \) such that \( |- t \in T \) in the simply typed lambda-calculus with booleans, pairs, and subtyping (pages ?? to 10 in the appendix)? If so, give an example. If not, explain why this cannot occur.

☒ Yes ☐ No

\[
\begin{align*}
t & = \lambda x: \text{Top}. \ x \\
T & = \text{Top}
\end{align*}
\]
11. **[Standard Only] (Subtyping)** (8 points)

For each of the following pairs of types, $S$ and $T$, mark the appropriate description of how they are ordered by the subtype relation. (In the last two, $\emptyset$ is the empty record type—a record type with no fields.)

11.1 $S = (\text{Top} \to \text{Bool}) \to (\text{Nat} \to \text{Top})$
   $T = \text{Top} \to (\text{Top} \to \text{Bool})$

- $\square S <: T$
- $\zeta T <: S$
- $\square$ equivalent (both $S <: T$ and $T <: S$)
- $\square$ unrelated (neither $S <: T$ nor $T <: S$)

11.2 $S = \{x: \text{Bool}, y: \text{Top}\}$
   $T = \{y: \text{Top} \to \text{Top}, x: \text{Top}\}$

- $\square S <: T$
- $\square T <: S$
- $\square$ equivalent
- $\zeta$ unrelated

11.3 $S = \{x: \emptyset, y: \text{Top}\}$
   $T = \{y: \emptyset\}$

- $\square S <: T$
- $\square T <: S$
- $\square$ equivalent
- $\zeta$ unrelated

11.4 $S = \emptyset$
   $T = \text{Top}$

- $\zeta S <: T$
- $\square T <: S$
- $\square$ equivalent
- $\square$ unrelated

12. **Subtyping** (15 points)

In this problem, we again consider the simply typed lambda-calculus with booleans, products, and subtyping (pages ?? to 10 in the appendix).
12.1 In this language, is there a type with infinitely many subtypes? (I.e., is there some type \( T \) such that the set of all \( S \) with \( S <: T \) is infinite?)

☑ Yes ☐ No

If Yes, give an example:

\textit{Example: Top}

12.2 Is there a type with infinitely many \textit{supertypes}? (I.e., is there some type \( S \) such that the set of all \( T \) with \( S <: T \) is infinite?)

☑ Yes ☐ No

If Yes, give an example:

\textit{Example: Top->Top}

12.3 Is there a type with only finitely many subtypes?

☑ Yes ☐ No

If Yes, give an example and list \textit{all} of its subtypes:

\textit{Example: Bool (its only subtype is itself)}

12.4 Is there a type with only finitely many supertypes?

☑ Yes ☐ No

If Yes, give an example and list \textit{all} of its supertypes:

\textit{Example: Top (and its only supertype is itself)}

12.5 Is there a term \( t \) that has infinitely many types in the empty context (i.e., such that the set of \( T \) with \( |- t \in T \) is infinite)?

☑ Yes ☐ No

If Yes, give an example:

\textit{Example: } \( \lambda x: \text{Top}. x \)
Formal definitions for Imp

Syntax

Inductive aexp : Type :=
  | ANum : nat -> aexp
  | AId : string -> aexp
  | APlus : aexp -> aexp -> aexp
  | AMinus : aexp -> aexp -> aexp
  | AMult : aexp -> aexp -> aexp.

Inductive bexp : Type :=
  | BTrue : bexp
  | BFalse : bexp
  | BEq : aexp -> aexp -> bexp
  | BLE : aexp -> aexp -> bexp
  | BNot : bexp -> bexp
  | BAnd : bexp -> bexp -> bexp.

Inductive com : Type :=
  | CSkip : com
  | CAss : string -> aexp -> com
  | CSeq : com -> com -> com
  | CIf : bexp -> com -> com -> com
  | CWhile : bexp -> com -> com.

Notation "'SKIP'" :=
  CSkip.
Notation "l '::=' a" :=
  (CAss l a).
Notation "c1 ;; c2" :=
  (CSeq c1 c2).
Notation "'WHILE' b 'DO' c 'END'" :=
  (CWhile b c).
Notation "'IFB' e1 'THEN' e2 'ELSE' e3 'FI'" :=
  (CIf e1 e2 e3).
Evaluation relation

Inductive ceval : com → state → state → Prop :=
  | E_Skip : forall st,
    SKIP / st \ st
  | E_Ass : forall st a1 n X,
    aeval st a1 = n →
    (X ::= a1) / st \ (update st X n)
  | E_Seq : forall c1 c2 st st' st'',
    c1 / st \ st' →
    c2 / st' \ st'' →
    (c1 ;; c2) / st \ st''
  | E_IfTrue : forall st st' b1 c1 c2,
    beval st b1 = true →
    c1 / st \ st' →
    (IFB b1 THEN c1 ELSE c2 FI) / st \ st'
  | E_IfFalse : forall st st' b1 c1 c2,
    beval st b1 = false →
    c2 / st \ st' →
    (IFB b1 THEN c1 ELSE c2 FI) / st \ st'
  | E_WhileEnd : forall b1 st c1,
    beval st b1 = false →
    (WHILE b1 DO c1 END) / st \ st
  | E_WhileLoop : forall st st' st'' b1 c1,
    beval st b1 = true →
    c1 / st \ st' →
    (WHILE b1 DO c1 END) / st' \ st'' →
    (WHILE b1 DO c1 END) / st \ st''

where "c1 '/' st '\\' st''" := (ceval c1 st st').

Program equivalence

Definition bequiv (b1 b2 : bexp) : Prop :=
  forall (st:state), beval st b1 = beval st b2.

Definition cequiv (c1 c2 : com) : Prop :=
  forall (st st' : state),
  (c1 / st \ st') <-> (c2 / st \ st').

Hoare triples

Definition hoare_triple (P:Assertion) (c:com) (Q:Assertion) : Prop :=
  forall st st', c / st \ st' → P st → Q st'.

Notation "{{ P }} c {{ Q }}" := (hoare_triple P c Q).
Implication on assertions

Definition assert_implies (P Q : Assertion) : Prop :=
   forall st, P st -> Q st.

Notation "P ->> Q" := (assert_implies P Q).

(ASCII ->>> is typeset as a hollow arrow in the rules below.)

Hoare logic rules

\[ \{ \text{assn\_sub X a Q} \} X := a \{ Q \} \]  (hoare\_asgn)

\[ \{ P \} \text{SKIP} \{ P \} \]  (hoare\_skip)

\[ \{ P \} c1 \{ Q \} \]
\[ \{ Q \} c2 \{ R \} \]
\[ \{ P \} c1 ;; c2 \{ R \} \]  (hoare\_seq)

\[ \{ P \wedge b \} c1 \{ Q \} \]
\[ \{ P \wedge \sim b \} c2 \{ Q \} \]
\[ \{ P \} \text{IFB b THEN c1 ELSE c2 FI} \{ Q \} \]  (hoare\_if)

\[ \{ P \wedge b \} c \{ P \} \]
\[ \{ P \} \text{WHILE b DO c END} \{ P \wedge \sim b \} \]  (hoare\_while)

\[ \{ P' \} c \{ Q' \} \]
\[ P \rightarrow P' \]
\[ Q' \rightarrow Q \]
\[ \{ P \} c \{ Q \} \]  (hoare\_consequence)

\[ \{ P' \} c \{ Q \} \]
\[ P \rightarrow P' \]
\[ \{ P \} c \{ Q \} \]  (hoare\_consequence\_pre)

\[ \{ P \} c \{ Q' \} \]
\[ Q' \rightarrow Q \]
\[ \{ P \} c \{ Q \} \]  (hoare\_consequence\_post)
STLC with booleans

Syntax

T ::= Bool 
    | T -> T 
    | \x:T. t
    | true
    | false
    | if t then t else t

t ::= x
    | t t 
    | if t then t else t

v ::= true
    | false

Small-step operational semantics

value v

\(\langle x:T.t \rangle v \Rightarrow [x:=v]t\)  (ST_AppAbs)

\(t_1 \Rightarrow t_1'\)  (ST_App1)

\(\langle x:T.t \rangle v \Rightarrow [x:=v]t\)

\(t_1 t_2 \Rightarrow t_1' t_2\)

value v1

\(t_2 \Rightarrow t_2'\)  (ST_App2)

\(v_1 t_2 \Rightarrow v_1 t_2'\)

\(\text{if true then } t_1 \text{ else } t_2 \Rightarrow t_1\)  (ST_IfTrue)

\(\text{if false then } t_1 \text{ else } t_2 \Rightarrow t_2\)  (ST_IfFalse)

\(t_1 \Rightarrow t_1'\)  (ST_If)

\(\text{if } t_1 \text{ then } t_2 \text{ else } t_3 \Rightarrow \text{if } t_1' \text{ then } t_2 \text{ else } t_3\)
Typing

Gamma x = T
------------------ (T_Var)
Gamma ⊢ x ∈ T

Gamma, x:T11 ⊢ t12 ∈ T12
----------------------------- (T_Abs)
Gamma ⊢ \x:T11.t12 ∈ T11->T12

Gamma ⊢ t1 ∈ T11->T12
Gamma ⊢ t2 ∈ T11
--------------------------- (T_App)
Gamma ⊢ t1 t2 ∈ T12

--------------------------- (T_True)
Gamma ⊢ true ∈ Bool

--------------------------- (T_False)
Gamma ⊢ false ∈ Bool

Gamma ⊢ t1 ∈ Bool  Gamma ⊢ t2 ∈ T  Gamma ⊢ t3 ∈ T
----------------------------------------------------- (T_If)
Gamma ⊢ if t1 then t2 else t3 ∈ T

Properties of STLC

Theorem preservation : forall t t' T,
empty ⊢ t ∈ T  =>
t ==> t'  =>
empty ⊢ t' ∈ T.

Theorem progress : forall t T,
empty ⊢ t ∈ T  =>
value t \/ exists t', t ==> t'.
STLC with products

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

\[
T ::= \ldots | T \times T | (t, t) | (v, v) \\
t ::= \ldots | t.\text{fst} | t.\text{snd} \\
v ::= \ldots
\]

Small-step operational semantics (added to STLC rules)

\[
t1 \Rightarrow t1' \\
(t1, t2) \Rightarrow (t1', t2) \quad \text{(ST_Pair1)}
\]

\[
t2 \Rightarrow t2' \\
(v1, t2) \Rightarrow (v1, t2') \quad \text{(ST_Pair2)}
\]

\[
t1 \Rightarrow t1' \\
t1.\text{fst} \Rightarrow t1'.\text{fst} \quad \text{(ST_Fst1)}
\]

\[
(v1, v2).\text{fst} \Rightarrow v1 \quad \text{(ST_FstPair)}
\]

\[
t1 \Rightarrow t1' \\
t1.\text{snd} \Rightarrow t1'.\text{snd} \quad \text{(ST_Snd1)}
\]

\[
(v1, v2).\text{snd} \Rightarrow v2 \quad \text{(ST_SndPair)}
\]

Typing (added to STLC rules)

\[
\Gamma \vdash t1 \in T1 \quad \Gamma \vdash t2 \in T2 \\
----------------------------- \quad \text{(T_Pair)}
\]

\[
\Gamma \vdash (t1, t2) \in T1 \times T2 \\
\Gamma \vdash t1 \in T11 \times T12 \quad \text{(T_Fst)}
\]

\[
\Gamma \vdash t1.\text{fst} \in T11 \\
\Gamma \vdash t1 \in T11 \times T12 \quad \text{(T_Snd)}
\]

\[
\Gamma \vdash t1.\text{snd} \in T12 \\
\]
STLC with Booleans and Subtyping

Extend the language from pages ?? to 9 with the type Top (terms and values remain unchanged):

T ::= ... |
    | Top

Add these rules that characterize the subtyping relation:

\[
\begin{align*}
S <: U & \quad U <: T \\ 
\hline
S <: T \\
\end{align*}
\]

(S_Trans)

\[
\begin{align*}
T <: T \\
\end{align*}
\]

(S_Refl)

\[
\begin{align*}
S <: \text{Top} \\
\end{align*}
\]

(S_Top)

\[
\begin{align*}
S1 <: T1 & \quad S2 <: T2 \\
\hline
S1 \ast S2 <: T1 \ast T2 \\
\end{align*}
\]

(S_Prod)

\[
\begin{align*}
T1 <: S1 & \quad S2 <: T2 \\
\hline
S1 \rightarrow S2 <: T1 \rightarrow T2 \\
\end{align*}
\]

(S_Arrow)

And add this to the typing relation:

\[
\begin{align*}
\Gamma \vdash t \in S & \quad S <: T \\
\hline
\Gamma \vdash t \in T \\
\end{align*}
\]

(T_Sub)
Big-step evaluation relation for STLC + Booleans

\[
\begin{align*}
\text{value } v \\
\text{-------} \\
v \text{ \ \ } v
\end{align*}
\]

\[
\begin{align*}
t1 \text{ \ \ } \\ \ \ \ x:T.t1' \\
t2 \text{ \ \ } v2 \\
[x:=v2]t1' \text{ \ \ } v \\
\text{------------------------------------------------- (E_App)}
\end{align*}
\]

\[
\begin{align*}
t0 \text{ \ \ } true \\
t2 \text{ \ \ } v
\end{align*}
\]

\[
\begin{align*}
\text{------------------------------------------ (E_IfTrue)}
\end{align*}
\]

\[
\begin{align*}
t0 \text{ \ \ } false \\
t2 \text{ \ \ } v
\end{align*}
\]

\[
\begin{align*}
\text{------------------------------------------ (E_IfFalse)}
\end{align*}
\]