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.

Mark the box of the track you are following (if you are not taking the exam as a WPE-I).

☐ Standard  ☐ Advanced

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!).
1 [Standard Track Only] Types of Coq terms (10 points)
For each Coq term below, give its type or write “ill-typed” if it has no type.

1.1 fun (X : Type) => fun (f : X -> X) => fun (a : X) => f (f (f a))

1.2 if true then 5 else false

1.3 fun x => x + x = 1

1.4 forall st, st = [ X := Y ] => st

1.5 (fun P => P \/ ~P) False
Coq programming (12 points)

For each type below, either give a term that has that type or write “uninhabited.”

2.1 $\forall X : \text{Type}, X \to (X \times \text{list } X)$

2.2 $\text{bool} \to \text{Prop}$

2.3 $\forall x : \text{bool}, \text{Prop}$

2.4 $\text{False} \to \text{bool}$

2.5 $2 + 2 = 5$
Loop invariants (9 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:

\[
\{ \{ P \} \} \text{ WHILE } b \text{ DO } c \text{ END } \{ \{ Q \} \}
\]

Specifically, you should decide whether Inv satisfies each of the three specific constraints from the Hoare rule for WHILE:

1. Implied by precondition: \( P \rightarrow Inv \)
2. Preserved by loop body (when loop guard true): \( \{ \{ Inv \land \neg b \} \} c \{ \{ Inv \} \} \)
3. Implies postcondition (when loop guard false): \( (Inv \land \neg b) \rightarrow Q \)

We call them “Implied by Pre,” “Preserved,” and “Implies Post” below, for brevity.

3.1 \[ \{ \{ X = 10 \} \} \]

\[ \text{WHILE } Y .< 3 \text{ DO} \]
\[ X := X .* 10;; \]
\[ Y := Y .+ 1 \]
\[ \text{END} \]
\[ \{ \{ X = 1000 \} \} \]

<table>
<thead>
<tr>
<th>Proposed Inv</th>
<th>Implied by Pre</th>
<th>Preserved</th>
<th>Implies Post</th>
</tr>
</thead>
<tbody>
<tr>
<td>exists (x : nat), X = 10 * x</td>
<td>☐</td>
<td>☐</td>
<td>☐</td>
</tr>
<tr>
<td>X &lt;= 1000</td>
<td>☐</td>
<td>☐</td>
<td>☐</td>
</tr>
<tr>
<td>X = 10 ^ Y</td>
<td>☐</td>
<td>☐</td>
<td>☐</td>
</tr>
</tbody>
</table>

3.2 \[ \{ \{ X = 10 /\ Y = 2 /\ Z = 0 \} \} \]

\[ \text{WHILE } !(X .= 0) \text{ DO} \]
\[ X ::= X .- Y;; \]
\[ Z ::= Z .+ 1 \]
\[ \text{END} \]
\[ \{ \{ Z = 5 \} \} \]

<table>
<thead>
<tr>
<th>Proposed Inv</th>
<th>Implied by Pre</th>
<th>Preserved</th>
<th>Implies Post</th>
</tr>
</thead>
<tbody>
<tr>
<td>10 = X + Z * Y /\ Y = 2</td>
<td>☐</td>
<td>☐</td>
<td>☐</td>
</tr>
<tr>
<td>X = Z * 2</td>
<td>☐</td>
<td>☐</td>
<td>☐</td>
</tr>
<tr>
<td>X = 10 + Y * Z /\ Y = 2</td>
<td>☐</td>
<td>☐</td>
<td>☐</td>
</tr>
</tbody>
</table>
Hoare logic with HAVOC (6 points)

Recall the language HImp from the Equiv chapter of Programming Language Foundations. Beginning from the standard Imp language, we add one new form of command

\[ c ::= \ldots \]

\[ \text{HAVOC } X \]

and one new clause to the ceval relation:

\[
| \text{E}_{\text{Havoc}} : \forall (st : \text{state}) (X : \text{string}) (n : \text{nat}), \\
\quad st =\left[ \text{HAVOC } X \right] \Rightarrow (X !-> n ; st) \\
\]

That is, HAVOC X nondeterministically sets X to any value. For example, suppose c is this program:

\[
\text{WHILE } !(X .\leq Y) \text{ DO} \\
\quad \text{HAVOC } X;; \\
\quad Y ::= Y .\+ 1 \\
\text{END}
\]

If we start c in a state st where X is already less than or equal to Y, then it will (deterministically) terminate in the same state. On the other hand, if we start c in a state where X is greater than Y then it is possible for c to terminate in \textit{any state } st' \textit{ where (1) the final value of X in } st' \textit{ is less than or equal to the final value of Y, (2) the final value of Y is greater than its starting value in } st, \textit{ and (3) the final values of the other variables are the same as their starting values.}

In the space below, write an appropriate Hoare Logic rule for reasoning about HAVOC statements. Your rule should have the form

\[
\text{-----------------------------} \\
\{\{ P \} \} \text{ HAVOC } X \{\{ Q \}\}
\]

for a suitable precondition P and postcondition Q.
Loop invariants with Havoc (15 points)

Now let's consider Hoare triples for the HImp language. For each triple below, find an invariant for the WHILE loop that will allow us to prove the triple. As usual, \( m \) and \( n \) are arbitrary (but fixed) values of type \( \text{nat} \).

5.1 \[
\{ X = m \land Y = n \} \\
\text{WHILE } \neg(X = Y) \text{ DO} \\
\quad \text{HAVOC } Z ;; \\
\quad \text{IFB } Z .<= Y \text{ THEN} \\
\quad \quad X ::= X .+ Z ;; \\
\quad \quad Y ::= Y .- Z \\
\quad \text{ELSE SKIP FI} \\
\quad \{ \text{Invariant goes here } \} \\
\text{END} \\
\{ 2 \times X = m + n \} 
\]

Invariant =

5.2 \[
\{ X <= 1 \} \\
\quad Y ::= 0 ;; \\
\quad \text{WHILE } Z .= 0 \text{ DO} \\
\quad \quad X ::= X .+ Y ;; \\
\quad \quad \text{HAVOC } Z \\
\quad \quad \{ \text{Invariant goes here } \} \\
\quad \text{END} \\
\{ X <= 1 \} 
\]

Invariant =

5.3 \[
\{ X = m \} \\
\quad \text{WHILE } X .<= m \text{ DO} \\
\quad \quad \text{HAVOC } Z ;; \\
\quad \quad \text{IFB } Z .= 0 \text{ THEN} \\
\quad \quad \quad X ::= X .+ X \\
\quad \quad \text{ELSE SKIP FI} \\
\quad \quad \{ \text{Invariant goes here } \} \\
\quad \text{END} \\
\{ X = 2 \times m \} 
\]

Invariant =
Refinement and HAVOC (12 points)

Suppose we define a refinement relation between HImp programs as follows:

Program \( c \) refines program \( d \) if \( st = [ c ] \Rightarrow st' \) implies \( st = [ d ] \Rightarrow st' \) for all \( st \) and \( st' \).

This is similar to the definition of cequiv that we studied in the Equiv chapter (repeated on page \[\] for reference), but an “iff” has been replaced by an implication (i.e., saying \( c_1 \) refines \( c_2 \) and \( c_2 \) refines \( c_1 \) is the same as saying that \( c_1 \) and \( c_2 \) are equivalent).

6.1 If \( c_1 \) refines \( c_2 \), does it follow that \( c_1 \) and \( c_2 \) terminate on the same set of initial memory states?

□ Yes □ No

6.2 \( c_1 = \text{HAVOC} X; \text{HAVOC} Y \)
\( c_2 = \text{HAVOC} X; Y ::= X \)
□ \( c_1 \) refines \( c_2 \) □ \( c_2 \) refines \( c_1 \) □ Both □ Neither

6.3 \( c_1 = \text{WHILE true DO SKIP END} \)
\( c_2 = \text{WHILE} X .:= 0 \text{ DO SKIP END} \)
□ \( c_1 \) refines \( c_2 \) □ \( c_2 \) refines \( c_1 \) □ Both □ Neither

6.4 \( c_1 = \text{HAVOC} X \)
\( c_2 = \text{HAVOC} X;; X ::= X + 1 \)
□ \( c_1 \) refines \( c_2 \) □ \( c_2 \) refines \( c_1 \) □ Both □ Neither

6.5 \( c_1 = \text{HAVOC} X;; \text{WHILE} X .> 0 \text{ DO X ::= X .- 1 END} \)
\( c_2 = X ::= 100;; \text{WHILE} X .> 0 \text{ DO HAVOC X END} \)
□ \( c_1 \) refines \( c_2 \) □ \( c_2 \) refines \( c_1 \) □ Both □ Neither

6.6 Let \( P \) be the set of HImp programs that mention the variables \( X, Y, \) and \( Z \) (or a subset of these, but no others). Does \( P \) have a minimal element in the refinement ordering—i.e., is there some \( p \in P \) such that \( p \) refines \( c \) for every \( c \in P \)?

□ Yes □ No

If so, then what is an example of such a minimal element?
6.7 Does the set $P$ have any maximal elements (i.e., any elements that are refined by every element of $P$)?

□ Yes  □ No

If so, then give an example.
[Advanced Track Only] Refinement and WHILE (10 points)

Give a detailed informal proof of the following theorem about the refinement relation from the previous problem. Please be sure to state your induction hypothesis *explicitly* at the beginning of each inductive case.

Theorem: If $c$ refines $d$, then $\text{WHILE } b \text{ DO } c \text{ END}$ refines $\text{WHILE } b \text{ DO } d \text{ END}$.

Proof:
8. **Extensionality** (7 points)

The Logic chapter of *Logical Foundations* introduced the axiom of *functional extensionality*.

8.1 State this axiom (in formal Coq notation, if possible, otherwise in words).

8.2 Briefly (1-3 sentences) explain how it is used in LF (e.g., in the treatment of total and partial maps in chapter Maps).
Recall the familiar Fibonacci function:

\[
\text{fib } x = \\
\quad \text{if } x = 0 \text{ then } 0 \\
\quad \quad \text{else if } x = 1 \text{ then } 1 \\
\quad \quad \text{else fib (x-2) + fib (x-1)}
\]

We saw in chapter MoreStlc how adding a \texttt{fix} primitive (see page 9 in the handout) to the STLC allows us to write such definitions in a more basic way.

Use \texttt{fix} to finish the following definition (without mentioning the identifier \texttt{fib}) in the STLC with numbers, booleans, and \texttt{fix}.

\[
\text{fib } = 
\]
10 Substitution as a relation (12 points)

Recall the definition of the substitution operation in the STLC extended with let (pages 6, 7) (but omitting base types such as booleans for brevity):

\[
\text{Fixpoint subst (x : string) (s : tm) (t : tm) : tm :=}
\begin{align*}
\text{match t with} \\
\text{let y y} \\
\text{if eqb_string x y then s else t} \\
\text{abs y T t1} \\
\text{abs y T (if eqb_string x y then t1 else (subst x s t1))} \\
\text{app t1 t2} \\
\text{app (subst x s t1) (subst x s t2)} \\
\text{tlet y t1 t2} \\
\text{tlet y (subst x s t1) (if eqb_string x y then t2 else (subst x s t2))}
\end{align*}
\]

This definition uses Coq’s Fixpoint facility to define substitution as a function. Suppose, instead, we wanted to define substitution as an inductive relation subst. We’ve begun the definition by providing the Inductive header and one of the constructors; your job is to fill in the rest of the constructors. (Your answer should have the property that subst x s t = t’ <-> subst x s t t’, for all s, x, t, and t’, but you do not need to prove it).

\[
\text{Inductive subst (x : id) (s : tm) : tm -> tm -> Prop :=}
\begin{align*}
\text{s_app : forall t1 t2 t1' t2',} \\
\text{subst x s t1' t1'} \\
\text{subst x s t2 t2'} \\
\text{subst x s (app t1 t2) (app t1' t2')} \\
\end{align*}
\]
Properties of reduction and typing (12 points)

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

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

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 \( t_1 \rightarrow t_2 \) to mean \( \text{stepsto } t_1 t_2 \) and \( |- t \in T \) to mean \( \text{has_type } t T \).

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

- Uniqueness of typing: If \( t \) is a term and \( T_1, T_2 \) are types such that \( |- t \in T_1 \) and \( |- t \in T_2 \), then \( T_1 = T_2 \).
- Determinism of reduction: If \( t \rightarrow t_1 \) and \( t \rightarrow t_2 \), then \( t_1 = t_2 \).
- Progress: If \( |- t \in T \), then either \( t \) is a value or else there is some \( t' \) such that \( t \rightarrow t' \).
- Preservation: If \( |- t \in T \) and \( t \rightarrow t' \), then \( |- t' \in T \).
- 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 (on the next page), 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 definitely break the corresponding property, check the box by “fails.” If the proposed change definitely cannot break the property, check the box next to “holds.” If this change 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.”
11.1 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

11.2 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

11.3 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

11.4 If we enlarge the reduction relation for well-typed terms (i.e., we add one or more new pairs $t \rightarrow t'$, 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
Progress and Preservation for STLC with Subtyping (12 points)

The syntax, operational semantics, and typing rules for the simply-typed lambda calculus with booleans, products, records, and subtyping are given in the Appendix (pages 6, 8, 9, 10).

For each variant below, indicate which of the properties of the original system remain true in the presence of this rule. If a property becomes false, give a counterexample.

12.1 Suppose that we add the following reduction rule:

\[
\begin{align*}
\text{false} & \rightarrow \lambda x: \text{Bool}. \ x \\
\end{align*}
\]

- Progress
  - □ Remains true □ Becomes false
  - Counterexample if false =

- Preservation
  - □ Remains true □ Becomes false
  - Counterexample if false =

12.2 Suppose instead that we add the following subtyping rule:

\[
\begin{align*}
\text{(S_BoolArg)} & \\
(\text{Bool} \rightarrow T) < \text{=} T \\
\end{align*}
\]

- Progress
  - □ Remains true □ Becomes false
  - Counterexample if false =

- Preservation
  - □ Remains true □ Becomes false
  - Counterexample if false =
Suppose, instead, that we consider a variant in which we add a new term \texttt{wrong} with the following reduction rule (and no typing rules):

\[
\begin{align*}
\text{--------------------- (ST}_{\text{WrongWrong}}) \\
\text{wrong} & \rightarrow \text{wrong wrong}
\end{align*}
\]

- **Progress**  
  Counterexample if false =

- **Preservation**  
  Counterexample if false =

\[\begin{array}{ll}
\text{□ Remains true} & \text{□ Becomes false} \\
\end{array}\]
Subtyping and typechecking (16 points)

13.1 empty |- (\p:T. \{x=p.x, y=p.y\}) \{x=true, y=unit, z=true\} \in \{x:Bool\}

(a) List all of the types that could replace the variable T if there are finitely many of them; otherwise write “Infinite”.

(b) What is the smallest type that T could be? If it doesn’t exist, write “None”.

(c) What is the largest type that T could be? If it doesn’t exist, write “None”.

13.2 empty |- \{x=(\p:Bool \rightarrow Bool. p true) (\b:Top. b), y=true, z=unit\} \in \{x:T\}

(a) List all of the types that could replace the variable T if there are finitely many of them; otherwise write “Infinite”.

(b) What is the smallest type that T could be? If it doesn’t exist, write “None”.

(c) What is the largest type that T could be? If it doesn’t exist, write “None”.
empty |- (\p:\{y:Unit, z:Unit\}. p.y) \in T

(a) What is the smallest type that T could be? If it doesn’t exist, write “None”.

(b) What is the largest type that T could be? If it doesn’t exist, write “None”.

empty |- (\p:Top. p) (\z:T. true) \in \{x:Top, y:Unit\} -> Bool

(a) What is the smallest type that T could be? If it doesn’t exist, write “None”.

(b) What is the largest type that T could be? If it doesn’t exist, write “None”.
Give a detailed informal proof of the following theorem about the subtype relation for the STLC with booleans and products (but no records) as defined on page 9. Please be sure to state your induction hypothesis explicitly at the beginning of each case where it is used.

Theorem: If $S_1 \rightarrow S_2 \subseteq T$ then either $T = \text{Top}$ or $T = T_1 \rightarrow T_2$ for some $T_1$ and $T_2$ with $T_1 \subseteq S_1$ and $S_2 \subseteq T_2$.

You may assume the following lemma (“Nothing Above Top”):

If $\text{Top} \subseteq T$, then $T = \text{Top}$.

Proof:
For Reference

Total maps

Definition total_map (A : Type) := string -> A.

Definition t_empty {A : Type} (v : A) : total_map A :=
  (fun _ => v).

Definition t_update {A : Type} (m : total_map A)
  (x : string) (v : A) :=
  (* eqb_string : string -> string -> bool *)
  fun x' => if eqb_string x x' then v else m x'.

Notation "x '!'->' v ';' m" := (t_update m x v)
Notation "a '!'->' x" := (t_update empty_st a x).

Useful facts about maps

Lemma t_apply_empty : forall (A : Type) (x : string) (v : A),
  (_ !-> v) x = v.

Lemma t_update_eq : forall (A : Type) (m : total_map A) x v,
  (x !-> v ; m) x = v.

Lemma t_update_neq : forall (A : Type) (m : total_map A) x1 x2 v,
  x1 <> x2 ->
  (x1 !-> v ; m) x2 = m x2.

Lemma t_update_shadow : forall (A : Type) (m : total_map A) x v1 v2,
  (x !-> v2 ; x !-> v1 ; m) = (x !-> v2 ; m).

Lemma t_update_same : forall (A : Type) (m : total_map A) x,
  (x !-> m x ; m) = m.

Lemma t_update_permute : forall (A : Type) (m : total_map A) v1 v2 x1 x2,
  x2 <> x1 ->
  (x1 !-> v1 ; x2 !-> v2 ; m)
  = (x2 !-> v2 ; x1 !-> v1 ; m).
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.

Infix ".+" := APlus (at level 50).
Infix ".-" := AMinus (at level 50).
Infix ".*" := AMult (at level 40).
Infix ".<=" := BLe (at level 70).
Infix ".=" := BEq (at level 70).
Infix ".&&" := BAnd (at level 80).
Notation "'!' b" := (BNot b) (at level 60).
Notation "'SKIP'" := CSkip.
Notation "'x '::=' a" := (CAss x a).
Notation "c1 ;; c2" := (CSeq c1 c2).
Notation "'WHILE' b 'DO' c 'END'" := (CWhile b c).
Notation "'IFB' b 'THEN' c1 'ELSE' c2 'FI'" := (CIf b c1 c2).

There are also implicit coercions from nat and string to aexp, and from bool to bexp.
Evaluation functions for expressions

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.
Evaluation relation for commands

Inductive ceval : com -> state -> state -> Prop :=
| E_Skip : forall st,
  st = [ SKIP ] => st
| E_Ass : forall st a1 n x,
  aeval st a1 = n ->
  st = [ x := a1 ] => (x !-> n ; st)
| E_Seq : forall c1 c2 st st’ st’’,
  st = [ c1 ] => st’ ->
  st’ = [ c2 ] => st’’ ->
  st = [ c1 ;; c2 ] => st’’
| E_IfTrue : forall st st’ b c1 c2,
  beval st b = true ->
  st = [ c1 ] => st’ ->
  st = [ IFB b THEN c1 ELSE c2 FI ] => st’
| E_IfFalse : forall st st’ b c1 c2,
  beval st b = false ->
  st = [ c2 ] => st’ ->
  st = [ IFB b THEN c1 ELSE c2 FI ] => st’
| E_WhileFalse : forall b st c,
  beval st b = false ->
  st = [ WHILE b DO c END ] => st
| E_WhileTrue : forall st st’ st’’ b c,
  beval st b = true ->
  st = [ c ] => st’ ->
  st’ = [ WHILE b DO c END ] => st’’ ->
  st = [ WHILE b DO c END ] => st’’

where "st = [ c ] => st’" := (ceval c st st’).

Program equivalence

Definition aequiv (a1 a2 : aexp) : Prop :=
forall (st : state), aeval st a1 = aeval st a2.

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),
(st = [ c1 ] => st’) <-> (st = [ c2 ] => st’).
Hoare Logic

Definition hoare_triple

\[
(P : \text{Assertion}) \ (c : \text{com}) \ (Q : \text{Assertion}) : \text{Prop} := \\
\forall st \ st', \ st = \{c\} \Rightarrow st' \rightarrow P \ st \rightarrow Q \ st'.
\]

Notation "\{P\} \ c \ \{Q\}" := (hoare_triple P c Q).

Definition assert_implies (P Q : \text{Assertion}) : \text{Prop} :=
\forall st, \ P \ st \rightarrow Q \ st.

Notation "P \rightarrow> Q" := (assert_implies P Q).

Hoare logic rules

\[
\begin{align*}
\text{\textbf{(hoare_asgn)}} \\
\{\{ \text{assn_sub} \ x \ a \ Q \}\} \ x := a \ \{\{ \ Q \}\}
\end{align*}
\]

\[
\begin{align*}
\text{\textbf{(hoare_skip)}} \\
\{\{ P \}\} \ \text{SKIP} \ \{\{ P \}\}
\end{align*}
\]

\[
\begin{align*}
\{\{ P \}\} \ c1 \ \{\{ Q \}\} \quad \{\{ Q \}\} \ c2 \ \{\{ R \}\}
\end{align*}
\]

\[
\begin{align*}
\text{\textbf{(hoare_seq)}} \\
\{\{ P \}\} \ c1 ; ; c2 \ \{\{ R \}\}
\end{align*}
\]

\[
\begin{align*}
\{\{ P \land b \}\} \ c1 \ \{\{ Q \}\} \quad \{\{ P \land \neg b \}\} \ c2 \ \{\{ Q \}\}
\end{align*}
\]

\[
\begin{align*}
\text{\textbf{(hoare_if)}} \\
\{\{ P \}\} \ \text{IFB} \ b \ \text{THEN} \ c1 \ \text{ELSE} \ c2 \ \text{FI} \ \{\{ Q \}\}
\end{align*}
\]

\[
\begin{align*}
\{\{ P \land b \}\} \ c \ \{\{ P \}\}
\end{align*}
\]

\[
\begin{align*}
\text{\textbf{(hoare_while)}} \\
\{\{ P \}\} \ \text{WHILE} \ b \ \text{DO} \ c \ \text{END} \ \{\{ P \land \neg b \}\}
\end{align*}
\]

\[
\begin{align*}
P \rightarrow> P' \quad \{\{ P' \}\} \ c \ \{\{ Q' \}\} \quad Q' \rightarrow> Q
\end{align*}
\]

\[
\begin{align*}
\text{\textbf{(hoare_consequence)}} \\
\{\{ P \}\} \ c \ \{\{ Q \}\}
\end{align*}
\]
STLC with booleans

Syntax

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

\[
t ::= x \\
| t \rightarrow t \\
| \text{v} \\
| \text{if } t \text{ then } t \text{ else } t \\
\]

Small-step operational semantics

\[
\text{value } v \\
\rightarrow \begin{array}{c}
(\text{ST_AppAbs}) \\
(\text{ST_App1}) \\
(\text{ST_App2}) \\
(\text{ST_IfTrue}) \\
(\text{ST_IfFalse}) \\
(\text{ST_If})
\end{array}
\]

Typing

\[
\text{Gamma} \ x = T \\
\rightarrow \begin{array}{c}
(\text{T_Var}) \\
(\text{T_Abs}) \\
(\text{T_True}) \\
(\text{T_App}) \\
(\text{T_FALSE}) \\
(\text{T_TRUE}) \\
(\text{T_BOOL})
\end{array}
\]

(Continued on next page.)
\[
\text{Gamma } \vdash t_1 \in \text{Bool} \quad \text{Gamma } \vdash t_2 \in T \quad \text{Gamma } \vdash t_3 \in T
\]

\[
\text{---------------------------------------- (T}_\text{If})\]

\[
\text{Gamma } \vdash \text{if } t_1 \text{ then } t_2 \text{ else } t_3 \in T
\]

Properties of STLC

Theorem preservation: \(\forall t \ t' \ T,\)
\[
\text{empty } \vdash t \in T \rightarrow \quad (t \rightarrow t') \rightarrow \quad \text{empty } \vdash t' \in T.
\]

Theorem progress: \(\forall t \ T,\)
\[
\text{empty } \vdash t \in T \rightarrow \quad \text{value } t \lor \text{exists } t', \ t \rightarrow t'.
\]

STLC with let

\[
t ::= \ldots
\]
\[
\quad | \text{let } x=t \text{ in } t \quad \text{let-binding}
\]

Small-step operational semantics (added to STLC rules)
\[
t_1 \rightarrow t_1'
\]
\[
\text{---------------------------------------- (ST}_\text{Let1)}\]
\[
\text{let } x=t_1 \text{ in } t_2 \rightarrow \text{let } x=t_1' \text{ in } t_2
\]
\[
\text{---------------------------------------- (ST}_\text{LetValue)}\]
\[
\text{let } x=v_1 \text{ in } t_2 \rightarrow [x:=v_1]t_2
\]

Typing (added to STLC rules)
\[
\text{Gamma } \vdash t_1 \in T_1 \quad x \rightarrow T_1; \text{ Gamma } \vdash t_2 \in T_2
\]
\[
\text{---------------------------------------- (T}_\text{Let)}\]
\[
\text{Gamma } \vdash \text{let } x=t_1 \text{ in } t_2 \in T_2
\]

Formal definition of STLC with let in Coq, for reference in exercise Substitution as a relation

Inductive \(\text{tm} : \text{Type} :\)
\[
\quad | \text{var} : \text{string} \rightarrow \text{tm}
\]
\[
\quad | \text{app} : \text{tm} \rightarrow \text{tm} \rightarrow \text{tm}
\]
\[
\quad | \text{abs} : \text{string} \rightarrow \text{ty} \rightarrow \text{tm} \rightarrow \text{tm}
\]
\[
\quad | \text{tlet} : \text{string} \rightarrow \text{tm} \rightarrow \text{tm} \rightarrow \text{tm}
\]
\[
\quad (* \text{i.e., } [\text{let } x=t_1 \text{ in } t_2] \ast)\]
STLC with products

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

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

Small-step operational semantics (added to STLC rules)

\[
t_1 \rightarrow t_1' \quad t_2 \rightarrow t_2' \\
----------- (ST_Pair1) \quad ----------- (ST_Pair2) \\
(t_1, t_2) \rightarrow (t_1', t_2) \quad (v_1, t_2) \rightarrow (v_1', t_2)
\]

\[
t_1 \rightarrow t_1' \\
----------- (ST_Fst1) \quad ----------- (ST_FstPair) \\
t_1.fst \rightarrow t_1'.fst \quad (v_1, v_2).fst \rightarrow v_1
\]

\[
t_1 \rightarrow t_1' \\
----------- (ST_Snd1) \quad ----------- (ST_SndPair) \\
t_1.snd \rightarrow t_1'.snd \quad (v_1, v_2).snd \rightarrow v_2
\]

Typing (added to STLC rules)

\[
\Gamma |- t_1 \in T_1 \quad \Gamma |- t_2 \in T_2 \\
----------------------------------------- (T_Pair) \\
\Gamma |- (t_1, t_2) \in T_1 \times T_2
\]

\[
\Gamma |- t \in T_1 \times T_2 \\
----------------------------------------- (T_Fst) \\
\Gamma |- t.fst \in T_1
\]

\[
\Gamma |- t \in T_1 \times T_2 \\
----------------------------------------- (T_Snd) \\
\Gamma |- t.snd \in T_2
\]
STLC with fix

Extend the STLC from pages 6 to 8 with a fixed-point operator:

\[ t ::= \ldots \]
\[ \mid \text{fix} \ t \] fixed-point operator

Small-step operational semantics (added to STLC rules)

\[
t1 \rightarrow t1'
\]
\[ \text{------------------ (ST_Fix1)} \]
\[ \text{fix } t1 \rightarrow \text{fix } t1' \]
\[ \text{----------------------------- (ST_FixAbs)} \]
\[ \text{fix } (\text{xf} : T1. t2) \rightarrow [\text{xf} := \text{fix } (\text{xf} : T1. t2)] \ t2 \]

Typing (added to STLC rules)

\[
\Gamma \vdash t1 \in T1\rightarrow T1 \quad \text{------------------ (T_Fix)}
\]
\[ \Gamma \vdash \text{fix } t1 \in T1 \]

STLC with Booleans, Products and Subtyping

Extend the language from pages 6 to 8 with the type Top (terms and values remain unchanged):

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

Add these rules that characterize the subtyping relation:

\[
S <: U \quad U <: T \quad \text{------------------ (S_Trans)} \]
\[ S <: T \quad T <: T \quad S <: \text{Top} \quad \text{----- (S_Ref1)} \quad \text{----- (S_Top)} \]
\[ S1 <: T1 \quad S2 <: T2 \quad \text{---------------- (S_Prod)} \]
\[ S1 \times S2 <: T1 \times T2 \quad T1 <: S1 \quad S2 <: T2 \quad \text{---------------- (S_Arrow)} \]
\[ S1 \rightarrow S2 <: T1 \rightarrow T2 \]

And add this to the typing relation:

\[
\Gamma \vdash t \in S \quad S <: T \quad \text{------------------ (T_Sub)}
\]
\[ \Gamma \vdash t \in T \]
Record Subtyping

Implicitly, labels of a record should not contain any repetition.

\[\begin{align*}
T & ::= \ldots & t & ::= \ldots & v & ::= \ldots \\
& \mid \{i_1:T_1, \ldots, i_n:T_n\} & & \mid \{i_1=t_1, \ldots, i_n=t_n\} & & \mid \{i_1=v_1, \ldots, i_n=v_n\} \\
& \mid t.i
\end{align*}\]

Small-step operational semantics (added to STLC rules)

\[ti \rightarrow ti' \quad (ST_{Rcd})\]

\[\begin{align*}
\{i_1=v_1, \ldots, i_m=v_m, i_{n}=t_i \ldots\} \\
\rightarrow \{i_1=v_1, \ldots, i_m=v_m, i_{n}=t'_i \ldots\}
\end{align*}\]

Typing (added to STLC rules)

\[\Gamma |- t_1 \in T_1 \ldots \Gamma |- t_n \in T_n \quad (T_{Rcd})\]

\[\Gamma |- \{i_1=t_1, \ldots, i_n=t_n\} \in \{i_1:T_1, \ldots, i_n:T_n\}\]

\[\Gamma |- t \in \{\ldots, i:T_i, \ldots\} \quad (T_{Proj})\]

Subtyping (added to STLC rules)

Adding fields gives a subtype (S_{RcdWidth}), we can apply subtyping to each field (S_{RcdDepth}), subtyping can reorder fields (S_{RcdPerm}).

\[n > m \quad (S_{RcdWidth})\]

\[\{i_1:T_1\ldots i_n:T_n\} <: \{i_1:T_1\ldots i_m:T_m\}\]

\[S_1 <: T_1 \ldots \quad S_n <: T_n \quad (S_{RcdDepth})\]

\[\{i_1:S_1\ldots i_n:S_n\} <: \{i_1:T_1\ldots i_n:T_n\}\]

\{i_1:S_1\ldots i_n:S_n\} is a permutation of \{j_1:T_1\ldots j_n:T_n\} \quad (S_{RcdPerm})

\[\{i_1:S_1\ldots i_n:S_n\} <: \{j_1:T_1\ldots j_n:T_n\}\]