1. (15 points) Put an X in the True or False box for each statement.

(a) If the Imp command \( c_1 \) is equivalent to \( c_2 \), then any program containing \( c_1 \) is equivalent to the same program with \( c_1 \) replaced by \( c_2 \).

\[ \checkmark \text{True} \quad \square \text{False} \]

(b) If \( \text{WHILE } b \text{ DO SKIP END} \) is equivalent to \( \text{SKIP} \), then \( b \) is equivalent to \( \text{false} \).

\[ \checkmark \text{True} \quad \square \text{False} \]

(c) Depending on what \( b \) is, either the program \( \text{IFB } b \text{ THEN } c_1 \text{ ELSE } c_2 \text{ FI} \) is equivalent to \( c_1 \) or it is equivalent to \( c_2 \).

\[ \square \text{True} \quad \checkmark \text{False} \]

(Counterexample: Let \( b \) be \( X > Y \), let \( c_1 \) be \( Z ::= 0 \), and let \( c_2 \) be \( Z ::= 1 \).)

(d) The Hoare triple \( \{X = 0\} \text{WHILE true DO } c \text{ END } \{X = 1\} \) is valid no matter what \( c \) is.

\[ \checkmark \text{True} \quad \square \text{False} \]

(e) If \( \{\text{True}\} \text{WHILE } b \text{ DO } c \text{ END } \{\text{False}\} \) is valid, then \( b \) is equivalent to \( \text{true} \).

\[ \checkmark \text{True} \quad \square \text{False} \]

(f) A principal disadvantage of the big-step style of operational semantics is that it does not cleanly distinguish programs that get “stuck” in the middle of their execution (no rule applies) from programs that loop or recurse infinitely.

\[ \checkmark \text{True} \quad \square \text{False} \]

(g) The \textbf{strong\_progress} theorem in the \textit{Smallstep} chapter (see page 7 in the appendix) implies that “every normal form is a value.”

\[ \checkmark \text{True} \quad \square \text{False} \]
(h) When we say that the `ceval` relation of Imp is *deterministic*, what we mean is, for every st and c, there is exactly one st’ such that c / st \ st’.

□ True ☒ False

(At *most* one, not exactly one.)

(i) If c / st ==>* c’ / st for every st, using the small-step semantics of Imp from the Smallstep chapter, then c and c’ are equivalent.

☒ True □ False

(j) The big-step evaluation of Imp programs can naturally be expressed in Coq as either an Inductive relation or a Fixpoint.

□ True ☒ False

2. (12 points) For each pair of (standard) Imp commands below, put an X next to “Equivalent” if the two programs are behaviorally equivalent (as defined by `cequiv` in the Appendix), or put an X next to “Counterexample:” and give a counterexample that leads to different behaviors.

(a) c₁ = IFB X > 10 THEN X ::= 0 ELSE SKIP FI c₂ = SKIP

□ Equivalent
☒ Counterexample: X = 500 (or X = n for any n larger than 10).

(b) c₁ = WHILE 1 <= X DO X ::= X + 1 END c₂ = WHILE 2 <= X DO X ::= X + 1 END

□ Equivalent
☒ Counterexample: st such that X = 1

(c) c₁ = X ::= Y;; Y ::= X c₂ = Y ::= X;; X ::= Y

□ Equivalent
☒ Counterexample: any st such that X <> Y

(d) c₁ = X ::= 0 ;; WHILE X <> Y DO X ::= X + 1 END c₂ = X ::= Y

☒ Equivalent
□ Counterexample:
(c) \( c_1 = \text{IF} X <> Y \text{ THEN} \)
\[ \text{WHILE true DO SKIP END} \]
\[ \text{ELSE} \]
\[ \text{SKIP} \]
\[ \text{FI} \]

\[ c_2 = \text{WHILE} X <> Y \text{ DO} \]
\[ X ::= X + 1;; \]
\[ Y ::= Y + 1;; \]
\[ \text{END} \]

3. (12 points) Recall that the assertion \( P \) appearing in the \texttt{hoare_while} rule is called the \textit{loop invariant}. For each loop shown below, check the box next to each assertion that is a valid loop invariant — that is, \( \{ P \land b \} c \{ P \} \), where \( c \) is the body of the loop. There may be zero or more than one of them.

(a) \texttt{WHILE} X<10 \texttt{ DO} X::=X-1 \texttt{ END}

- \( X < 10 \)
- \( X = 10 \)
- \( X = 0 \)
- \( X <> 0 \)
- \( \text{False} \)
- \( \text{True} \)

(b) \texttt{WHILE} X>10 \texttt{ DO} Y::=Y+1 \texttt{ END}

- \( X < 10 \)
- \( X > 10 \)
- \( Y < 10 \)
- \( Y > 10 \)
- \( \text{False} \)
- \( \text{True} \)

(c) \texttt{WHILE} Y>0 \texttt{ DO} Y::=Y-1;; X::=X+1 \texttt{ END}

- \( X > 10 \)
- \( Y > 10 \)
- \( X + Y = Z \)
- \( Z + Y < X \)
- \( X <> Z \)
- \( Z = 3 \)
4. (9 points) In this problem, we will consider a simplified nondeterministic variant of Imp. There are three changes:

- The language contains a new construct `CHOOSE c1 OR c2 END`, which can execute as either `c1` or `c2`. Because of this nondeterministic construct, there can be many possible branches of execution, with different final outcomes.

- Additionally, the language contains two new special commands, `ASSERT b` and `ASSUME b`. Both commands do nothing if `b` evaluates to true; however, they differ in how they handle the error case when `b` is false:
  - If an `ASSERT` command fails, it causes the program to raise an error and terminate.
  - If an `ASSUME` command fails, the program (in this particular branch) fails to evaluate at all. In other words, the evaluation relation is not defined in states where the `ASSUME` command fails.

Intuitively, `ASSERT b` is a way to signal an error in branches of execution where `b` is found to be false, whereas `ASSUME b` is a way to cleanly discard branches of execution where `b` is found to be false (without signaling any error or terminating in any state).

- Finally, to keep things a bit simpler, we remove the commands `IFB` and `WHILE` from Imp.

Here is the syntax of our language:

```ocaml
Inductive com : Type :=
| CSkip : com
| CAss : string -> aexp -> com
| CSeq : com -> com -> com
| CBranch : com -> com -> com
| CAssert : bexp -> com
| CAssume : bexp -> com.
```

Notation "'SKIP'" := CSkip.
Notation "'x '::=' a" := (CAss x a).
Notation "c1 ;; c2" := (CSeq c1 c2).
Notation "'CHOOSE' c1 'OR' c2 'END'" := (CBranch c1 c2).
Notation "'ASSERT' b" := (CAssert b).
Notation "'ASSUME' b" := (CAssume b).

To define the evaluation relation, we need to distinguish between two kinds of termination: (1) normal termination with a final state, and (2) raising an error. We introduce the `result` data type for this purpose:

```ocaml
Inductive result : Type :=
| RNormal : state -> result
| RError : result.
```
We write $c / st \ \backslash \ r$ to mean that running command $c$ in state $st$ may terminate (in some branch) with result $r$, where $r$ is either $RNormal \ st'$, with $st'$ an ending state, or $RError$, denoting an error. For any command and initial state:

- If the command is $SKIP$, then the command terminates normally with the same state.
- If the command is an assignment, then it terminates normally, and the result state is updated with the new assignment to the variable.
- If the command is a sequence $c1 ;; c2$, then in branches where $c1$ raises an error, we raise an error. In branches where $c1$ terminates, we evaluate $c2$.
- If the command is a nondeterministic choice $CHOOSE \ c1 \ OR \ c2 \ END$, then we evaluate either $c1$ or $c2$ – the result of either branch is possible.
- For the command $ASSERT \ b$, if $b$ evaluates to true then we terminate normally with the same state. If $b$ evaluates to false, then we raise an error.
- For the command $ASSUME \ b$, if $b$ evaluates to true then we terminate normally with the same state. We do not have a case for when $b$ evaluates to false, because we do not want any result at all.

For example, if we run the command

```
CHOOSE
  X ::= 1
OR
  X ::= 3
END
```

from an all-zero initial state, then there are two possible results, both tagged $RNormal$: one where $X = 1$ and one where $X = 3$. On the other hand,

```
CHOOSE
  X ::= 1
OR
  X ::= 3
END;;
ASSUME X < 2
```

has only one possible final state — the one where $X = 1$ (again tagged $RNormal$) — while

```
CHOOSE
  X ::= 1
OR
  X ::= 3
END;;
ASSERT X < 2
```
has two results: a normal state where $X = 1$ and RError.

Based on the above description and example, complete the formal definition of the ceval relation on the following page. (Remember that the result should be either RNormal st or RError, with no associated state!)

\[
\text{Inductive ceval : com -> state -> result -> Prop :=}
\]
\[
| \text{E_Skip : forall st,} \\
| \text{SKIP / st \ \ RNormal st} \\
| \text{E_Ass : forall st a1 n x,} \\
| \text{aeval st a1 = n ->} \\
| \text{(x ::= a1) / st \ \ RNormal (st & \{ x --> n \})} \\
| \text{E_SeqNormal : forall c1 c2 st st' r,} \\
| \text{c1 / st \ \ RNormal st' ->} \\
| \text{c2 / st' \ \ r ->} \\
| \text{(c1 ;; c2) / st \ \ r} \\
| \text{E_SeqError : forall c1 c2 st,} \\
| \text{c1 / st \ \ RError ->} \\
| \text{(c1 ;; c2) / st \ \ RError} \\
| \text{E_BranchLeft : forall c1 c2 st,} \\
| \text{c1 / st \ \ r ->} \\
| \text{CHOOSE c1 OR c2 END \ \ r} \\
| \text{E_BranchRight : forall c1 c2 st,} \\
| \text{c2 / st \ \ r ->} \\
| \text{CHOOSE c1 OR c2 END \ \ r} \\
| \text{E_AssertTrue : forall st b,} \\
| \text{beval st b = true ->} \\
| \text{(ASSERT b) / st \ \ RNormal st} \\
| \text{E_AssertFalse : forall st b,} \\
| \text{beval st b = false ->} \\
| \text{(ASSERT b) / st \ \ RError} \\
| \text{E_Assume : forall st b,} \\
| \text{beval st b = true ->} \\
| \text{(ASSUME b) / st \ \ RNormal st}
\]

5. (15 points) (Continuation of previous problem.)

(a) What are the possible final results of the following programs, when started from an arbitrary initial state? (Put an X in the box next to each result that is possible.)

(i) \text{ASSERT (X = 1)}

\[\begin{array}{c}\checkmark \ \text{RError} \ \ \square \ \text{RNormal} \{X --> 0\} \ \ \checkmark \ \text{RNormal} \{X --> 1\} \ \ \square \ \text{RNormal} \{X --> 2\}\end{array}\]

(ii) \text{ASSUME (X = 1)}

\[\begin{array}{c}\square \ \text{RError} \ \ \square \ \text{RNormal} \{X --> 0\} \ \ \checkmark \ \text{RNormal} \{X --> 1\} \ \ \square \ \text{RNormal} \{X --> 2\}\end{array}\]
(iii) ASSUME (X = 1);; ASSERT (X = 1)
    □ RError □ RNormal {X --&gt; 0} □ RNormal {X --&gt; 1} □ RNormal {X --&gt; 2}

(iv) CHOOSE X ::= 1 OR X ::= 2 END;; ASSUME (X = 2)
    □ RError □ RNormal {X --&gt; 0} □ RNormal {X --&gt; 1} □ RNormal {X --&gt; 2}

(v) CHOOSE X ::= 1 OR X ::= 2 END;; ASSERT (X = 2)
    □ RError □ RNormal {X --&gt; 0} □ RNormal {X --&gt; 1} □ RNormal {X --&gt; 2}

(vi) CHOOSE
    ASSUME (X = 1);;
    X ::= 0
    OR
    ASSUME !(X = 1);;
    X ::= 1
    END
    □ RError □ RNormal {X --&gt; 0} □ RNormal {X --&gt; 1} □ RNormal {X --&gt; 2}

(b) In this language, we can implement the IF command as a derived construct. Specifically, we implement IFB b THEN c1 ELSE c2 END as the following:

CHOOSE
    ASSUME b;;
    c1
    OR
    ASSUME !(X = 1);;
    c2
END

We could instead have tried to implement IF with ASSERT instead of with ASSUME, namely:

CHOOSE
    ASSERT b;;
    c1
    OR
    ASSERT !(X = 1);;
    c2
END

Briefly explain why the latter would not be a good idea.

Answer: The command IFB X=0 THEN SKIP ELSE SKIP FI does not yield an error when executed, so a correct translation in terms of CHOOSE should not either. If we translate the IFB following the original suggestion (using ASSUME), then exactly one branch will be able to proceed and the other will get thrown away (with no failure). If we change the ASSUMEs to ASSERTs, the behavior will be to either behave like the THEN or ELSE branch, as appropriate, or (nondeterministically) to fail.
The next step is to develop Hoare rules for our language. We have to adapt the definition of a Hoare triple to accommodate error results. The new meaning of \{ P \} c \{ Q \} is: If c is started in a state satisfying P, then, after every possible execution, the result is a normal termination (not an error) with a final state satisfying Q. Formally:

\[
\text{Definition hoare_triple (P:Assertion) (c:com) (Q:Assertion) : Prop := forall st r, P st \rightarrow c / st \setminus r \rightarrow (exists st', r = \text{RNormal st'} \setminus Q st').}
\]

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

(c) Show that the commands ASSERT and ASSUME behave differently by giving an example of a precondition P, a boolean expression b, and a postcondition Q, such that \{ P \} ASSERT b \{ Q \} is invalid but \{ P \} ASSUME b \{ Q \} is valid.

\text{Answer: Take } P = \text{true}, b = \text{false}, \text{and } Q = \text{false}. \text{Then}
\[
\{ \text{true} \} \text{ ASSERT false } \{ \text{false} \}
\]
is not valid because it raises an error, but
\[
\{ \text{true} \} \text{ ASSUME false } \{ \text{false} \}
\]
is valid because there are no possible evaluations.

(d) With this extended Imp language, the old commands SKIP, assignment, and sequencing satisfy the same Hoare rules as before. The rule for CHOOSE is as follows:

\[
\begin{align*}
\{ P \} \text{ c1 } \{ Q \} \\
\{ P \} \text{ c2 } \{ Q \} \\
\{ P \} \text{ CHOOSE c1 OR c2 FI } \{ Q \} \end{align*}
\] (hoare_branch)

Complete the following Hoare rules (using informal notation, as above) for ASSERT and ASSUME. You do not need to prove your rules. For full credit, make sure your rules are not only valid, but as general as possible.

\text{Answer:}
\[
\{ Q \land b \} \text{ ASSERT b } \{ Q \} \quad \text{(hoare_assert)}
\]
\[
\{ P \} \text{ ASSUME b } \{ P \land b \} \quad \text{(hoare_assume)}
\]

Alternatively:
\[
\{ b \rightarrow Q \} \text{ ASSUME b } \{ Q \} \quad \text{(hoare_assume)}
\]
6. (5 points) The greatest common divisor (GCD) of two numbers \( m \) and \( n \), written \( \text{gcd}(m,n) \), is the largest number that evenly divides both \( m \) and \( n \). The familiar Euclidean algorithm is an effective recursive algorithm for computing the GCD:

\[
\text{gcd}(m,n) = \begin{cases} 
  m & m = n \\
  \text{gcd}(m-n,n) & m > n \\
  \text{gcd}(m,n-m) & m < n
\end{cases}
\]

(Coq won’t accept this as a definition because it will fail to pass the termination checker. It isn’t very hard to reformulate it in a way that makes Coq happy, but we’ll stick with this one here, since it’s a bit easier to look at.)

The following Imp program computes the GCD of \( X \) and \( Y \) and leaves it in the variable \( X \).

```
WHILE !(X = Y) DO
  IFB Y <= X THEN
    X ::= X - Y
  ELSE
    Y ::= Y - X
  FI
END
```

Suppose we want to prove the Hoare triple \( \{ \{ X = m \land Y = n \} \} \ gcd\_prog \{ \{ X = \text{gcd}(m,n) \} \} \), where \( \text{gcd}\_prog \) is the Imp program above.

Write down a suitable loop invariant \( P \) for this proof (i.e., one that is both valid and useful for writing down a decorated program validating the above Hoare triple).

Answer:

\( \text{gcd}(m,n) = \text{gcd}(X,Y) \)

7. [Standard Only] (12 points) On the next page, add appropriate annotations to the GCD program in the provided spaces to show that the Hoare triple given by the outermost pre- and post-conditions is valid. Please be completely precise and pedantic in the way you apply the Hoare rules — i.e., write out assertions in exactly the form given by the rules (rather than logically equivalent ones). The provided blanks have been constructed so that, if you work backwards from the end of the program, you should only need to use the rule of consequence in the places indicated with \(-\rightarrow\).

The implication steps in your decoration may rely (silently) on all the usual rules of natural-number arithmetic. You may also use the following equations involving the mathematical specification \( \text{gcd} \), but you should indicate (in the margin next to the \(-\rightarrow\) symbol) when you are using one of them in an application of the rule of consequence.

\begin{align*}
\text{Lemma 1 (gcd_gt).} \forall x \; y, \; x > y & \rightarrow \text{gcd}(x,y) = \text{gcd}(x-y,y) \\
\text{Lemma 2 (gcd_lt).} \forall x \; y, \; x < y & \rightarrow \text{gcd}(x,y) = \text{gcd}(x,y-x) \\
\text{Lemma 3 (gcd_eq).} \forall n, \; \text{gcd}(n,n) & = n
\end{align*}
The rules of Hoare logic and the rules for well-formed decorated programs can be found on pages 3 and 4 of the appendix.

You may assume that m and n are both positive.

\[
\begin{align*}
&\{ X = m \land Y = n \} \Rightarrow \{ \gcd m n = \gcd X Y \} \\
&\text{WHILE } !(X = Y) \text{ DO} \\
&\quad \{ \gcd m n = \gcd X Y \land X \neq Y \} \\
&\quad \text{IF } Y \leq X \\
&\quad \quad \{ \gcd m n = \gcd X Y \land X \neq Y \land Y \leq X \} \Rightarrow \text{(using gcd_gt)} \\
&\quad \quad \{ \gcd m n = \gcd (X - Y) Y \} \\
&\quad \quad X ::= X - Y \\
&\quad \{ \gcd m n = \gcd X Y \} \\
&\quad \text{ELSE} \\
&\quad \quad \{ \gcd m n = \gcd X Y \land X \neq Y \land \neg (Y \leq X) \} \Rightarrow \text{(using gcd_lt)} \\
&\quad \quad \{ \gcd m n = \gcd X (Y - X) \} \\
&\quad \quad Y ::= Y - X \\
&\quad \{ \gcd m n = \gcd X Y \} \\
&\quad \FI \\
&\{ \gcd m n = \gcd X Y \} \\
&\text{END} \\
&\{ \gcd m n = \gcd X Y \land \neg (X \neq Y) \} \Rightarrow \text{(using gcd_eq)} \\
&\{ \gcd m n = X \} \\
\end{align*}
\]

8. **[Advanced Only]** (12 points) Recall the simple expression language with plus (P) and numeric constants (C), which we introduced in Smallstep.v. (It is reproduced on page 8 of the Appendix, with both big-step and small-step semantics.) The following theorem captures the intuition that “big-step reduction implies small-step.”

**Theorem:** For all terms t and natural numbers n, if t \ n then t ==>* C n.

In the space below, write a careful proof of this theorem in English. Your proof may refer to the three lemmas on page 9 in the appendix.

**Answer:** Proof: By induction on a derivation of t \ n.

- Suppose the final rule used to show t \ n is E(Const). Then t = C n = v. We must show C n ==>* C n. This holds by multi_refl.

- Suppose the final rule used to show t \ n is E_Plus. Then t = P t1 t2, and we know that t1 \ C n1 and t2 \ C n2 for some n1 and n2. The IH tells us that t1 ==>* C n1 and t2 ==>* C n2. We must show that P t1 t2 ==>* C (n1 + n2).

First, notice that

\[
P \ t1 \ t2 ==>* \ P \ (C \ n1) \ t2
\]
by multistep_congr_1 and the =>* derivation for \( t_1 \). Observing that \( C \, n_1 \) is a value, we also notice

\[
P (C \, n_1) \, t_2 \Rightarrow说明书* P (C \, n_1) \, (C \, n_2)
\]

by multistep_congr_2 and the =>* derivation for \( t_2 \). It’s also easy to see by ST_PlusConstConst that

\[
P (C \, n_1) \, (C \, n_2) \Rightarrow C \, (plus \, n_1 \, n_2)
\]

and so, by multi_step and multi_refl, that the same is true for =>*. We can now use transitivity of =>* to stitch these derivations, proving

\[
P \, t_1 \, t_2 \Rightarrow* C \, (plus \, n_1 \, n_2).
\]
For Reference

Formal definitions for Imp

Syntax

Inductive aexp : Type :=
| ANum : nat -> aexp
| AId : id -> 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 : id -> aexp -> com
| CSeq : com -> com -> com
| CIf : bexp -> com -> com -> com
| CWhile : bexp -> com -> 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 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),
(c1 / st \ st’) <-> (c2 / st \ st’).
Hoare triples

Definition hoare_triple (P:Assertion) (c:com) (Q:Assertion) : Prop :=
for all 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 :=
for all 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

\[ \{\text{assign} \, X \, a \, Q\} \, X := a \, \{ \{ Q \} \} \quad \text{(hoare_asgn)} \]

\[ \{ \{ P \} \} \text{SKIP} \{ \{ P \} \} \quad \text{(hoare_skip)} \]

\[ \{ \{ P \} \} \, c \, \{ \{ Q \} \} \]
\[ \{ \{ Q \} \} \, c_2 \, \{ \{ R \} \} \]
\[ \{ \{ P \} \} \, c_1 \, ; \, c_2 \, \{ \{ R \} \} \quad \text{(hoare_seq)} \]

\[ \{ \{ P \wedge b \} \} \, c_1 \, \{ \{ Q \} \} \]
\[ \{ \{ P \wedge \neg b \} \} \, c_2 \, \{ \{ Q \} \} \]
\[ \{ \{ P \} \} \text{IFB} \, b \, \text{THEN} \, c_1 \, \text{ELSE} \, c_2 \, \text{FI} \, \{ \{ Q \} \} \quad \text{(hoare_if)} \]

\[ \{ \{ P \wedge b \} \} \, c \, \{ \{ P \} \} \]
\[ \{ \{ P \} \} \text{WHILE} \, b \, \text{DO} \, c \, \text{END} \, \{ \{ P \wedge \neg b \} \} \quad \text{(hoare_while)} \]

\[ \{ \{ P' \} \} \, c \, \{ \{ Q \} \} \]
\[ \quad \text{P} \rightarrow \text{P}' \]
\[ \{ \{ P \} \} \, c \, \{ \{ Q \} \} \quad \text{(hoare_consequence_pre)} \]

\[ \{ \{ P \} \} \, c \, \{ \{ Q' \} \} \]
\[ \quad \text{Q}' \rightarrow \text{Q} \]
\[ \{ \{ P \} \} \, c \, \{ \{ Q \} \} \quad \text{(hoare_consequence_post)} \]
Decorated programs

(a) SKIP is locally consistent if its precondition and postcondition are the same:

\[
\begin{align*}
\{\{ P \}\} \\
\text{SKIP} \\
\{\{ P \}\}
\end{align*}
\]

(b) The sequential composition of \(c_1\) and \(c_2\) is locally consistent (with respect to assertions \(P\) and \(R\)) if \(c_1\) is locally consistent (with respect to \(P\) and \(Q\)) and \(c_2\) is locally consistent (with respect to \(Q\) and \(R\)):

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

(c) An assignment is locally consistent if its precondition is the appropriate substitution of its postcondition:

\[
\begin{align*}
\{\{ P \ [X |-> a] \}\} \\
X ::= a \\
\{\{ P \}\}
\end{align*}
\]

(d) A conditional is locally consistent (with respect to assertions \(P\) and \(Q\)) if the assertions at the top of its "then" and "else" branches are exactly \(P \land b\) and \(P \land \neg b\) and if its "then" branch is locally consistent (with respect to \(P \land b\) and \(Q\)) and its "else" branch is locally consistent (with respect to \(P \land \neg b\) and \(Q\)):

\[
\begin{align*}
\{\{ P \}\} \\
\text{IFB} \ b \ \text{THEN} \\
\{\{ P \land b \}\} \\
\text{c1} \\
\{\{ Q \}\} \\
\text{ELSE} \\
\{\{ P \land \neg b \}\} \\
\text{c2} \\
\{\{ Q \}\} \\
\text{FI} \\
\{\{ Q \}\}
\end{align*}
\]
(e) A while loop with precondition $P$ is locally consistent if its postcondition is $P \land \neg b$ and if the pre- and postconditions of its body are exactly $P \land b$ and $P$:

```
{{ P }}
WHILE b DO
    {{ P \land b }}
    c1
    {{ P }}
END
{{ P \land \neg b }}
```

(f) A pair of assertions separated by $-\gg$ is locally consistent if the first implies the second (in all states):

```
{{ P }} -\gg
{{ P' }}
```

Relations

Definition relation (X: Type) := X->X->Prop.

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 x y ->
    multi R y z ->
    multi R x z.

Notation " t '==>*' t' " := (multi step t t').

Definition normal_form {X:Type} (R:relation X) (t:X) : Prop :=
  ~ exists t', R t t'.

5
Small Step Semantics

Reserved Notation " t '/' st '==>a' t' ".

Inductive astep : state -> aexp -> aexp -> Prop :=
  | AS_Id : forall st i,
    AId i / st ==>a ANum (st i)
  | AS_Plus : forall st n1 n2,
    APlus (ANum n1) (ANum n2) / st ==>a ANum (n1 + n2)
  | AS_Plus1 : forall st a1 a1' a2,
    a1 / st ==>a a1' ->
    (APlus a1 a2) / st ==>a (APlus a1' a2)
  | AS_Plus2 : forall st v1 a2 a2',
    aval v1 ->
    a2 / st ==>a a2' ->
    (APlus v1 a2) / st ==>a (APlus v1 a2')
  | AS_Minus : forall st n1 n2,
    (AMinus (ANum n1) (ANum n2)) / st ==>a (ANum (minus n1 n2))
  | AS_Minus1 : forall st a1 a1' a2,
    a1 / st ==>a a1' ->
    (AMinus a1 a2) / st ==>a (AMinus a1' a2)
  | AS_Minus2 : forall st v1 a2 a2',
    aval v1 ->
    a2 / st ==>a a2' ->
    (AMinus v1 a2) / st ==>a (AMinus v1 a2')
  | AS_Mult : forall st n1 n2,
    (AMult (ANum n1) (ANum n2)) / st ==>a (ANum (mult n1 n2))
  | AS_Mult1 : forall st a1 a1' a2,
    a1 / st ==>a a1' ->
    (AMult a1 a2) / st ==>a (AMult a1' a2)
  | AS_Mult2 : forall st v1 a2 a2',
    aval v1 ->
    a2 / st ==>a a2' ->
    (AMult v1 a2) / st ==>a (AMult v1 a2')
where " t '/' st '==>a' t' " := (astep st t t').

(And similarly for bstep.)
Reserved Notation " t '/' st '==>' t' '/' st' ".

Inductive cstep : (com * state) -> (com * state) -> Prop :=
  | CS_AssStep : forall st i a a',
    a / st ==>a a' ->
    (i ::= a) / st ==> (i ::= a') / st
  | CS_Ass : forall st i n,
    (i ::= (ANum n)) / st ==> SKIP / (t_update st i n)
  | CS_SeqStep : forall st c1 c1' st' c2,
    c1 / st ==> c1' / st' ->
    (c1 ;; c2) / st ==> (c1' ;; c2) / st'
  | CS_SeqFinish : forall st c2,
    (SKIP ;; c2) / st ==> c2 / st
  | CS_IfTrue : forall st c1 c2,
    IFB BTrue THEN c1 ELSE c2 FI / st ==> c1 / st
  | CS_IfFalse : forall st c1 c2,
    IFB BFalse THEN c1 ELSE c2 FI / st ==> c2 / st
  | CS_IfStep : forall st b b' c1 c2,
    b / st ==>b b' ->
    IFB b THEN c1 ELSE c2 FI / st
    ==> (IFB b' THEN c1 ELSE c2 FI) / st
  | CS_While : forall st b c1,
    (WHILE b DO c1 END) / st
    ==> (IFB b THEN (c1 ;; (WHILE b DO c1 END)) ELSE SKIP FI) / st
where " t '/' st '==>' t' '/' st' " := (cstep (t,st) (t',st')).

Theorem strong_progress : forall t,
  value t \/ (exists t', t ==> t').
Simple expression language

Inductive tm : Type :=
  | C : nat -> tm
  | P : tm -> tm -> tm.

Inductive value : tm -> Prop :=
  | v_const : forall n, value (C n).

Inductive eval : tm -> nat -> Prop :=
  | E_Const : forall n,
    C n \ n
  | E_Plus : forall t1 t2 n1 n2,
    t1 \ n1 ->
    t2 \ n2 ->
    P t1 t2 \ (n1 + n2)

  where " t '\' n " := (eval t n).

Inductive step : tm -> tm -> Prop :=
  | ST_PlusConstConst : forall n1 n2,
    P (C n1) (C n2) ==> C (n1 + n2)
  | ST_Plus1 : forall t1 t1' t2,
    t1 ==> t1' ->
    P t1 t2 ==> P t1' t2
  | ST_Plus2 : forall n1 t2 t2',
    t2 ==> t2' ->
    P (C n1) t2 ==> P (C n1) t2'

  where " t '==>' t' " := (step t t').
Lemmas for multistep relation

Lemma multistep_congr_1 : forall t1 t1' t2,
   t1 ===> t1' ->
   P t1 t2 ===> P t1' t2.

Lemma multistep_congr_2 : forall t1 t2 t2',
   value t1 ->
   t2 ===> t2' ->
   P t1 t2 ===> P t1 t2'.

Lemma multi_trans : forall (X:Type) (R: relation X) (x y z : X),
   multi R x y ->
   multi R y z ->
   multi R x z.