1. (5 points) Recall the definition of equivalence for while programs:

\[
\text{Definition cequiv (c1 c2 : com) : Prop :=} \\
\quad \text{forall (st st' : state), (c1 / st \rightarrow st') \leftrightarrow (c2 / st \rightarrow st').}
\]

Which of the following pairs of programs are equivalent? Write “yes” or “no” for each one. (Where it appears, \(a\) is an arbitrary aexp — i.e., you should write “yes” only if the two programs are equivalent for every \(a\).)

(a) \(X ::= A4\)
and
\(Y ::= A2 \; +++ \; A2;\)
\(X ::= Y\)

*Answer: No*

(b) \(X ::= a;\)
\(Y ::= a\)
and
\(Y ::= a;\)
\(X ::= a\)

*Answer: No*

(c) \(\text{while BTrue do (X := !X +++ 1)}\)
and
\(X ::= !X +++ 1\)

*Answer: No*

(d) \(\text{while BTrue do (X := !X +++ 1)}\)
and
\(\text{while BTrue do (X := !X -- 1)}\)

*Answer: Yes*

(e) \(\text{while BFalse do (X := !X +++ 1)}\)
and
\(\text{skip}\)

*Answer: Yes*

2. (5 points) Is this claim...
Claim: Suppose the command $c$ is equivalent to $c; c$. Then, for any $b$, the command

$$\text{while } b \text{ do } c$$

is equivalent to

$$\text{testif } b \text{ then } c \text{ else skip.}$$

... true or false? Briefly explain.

Answer: False. If $b$ evaluates to true and $c$ does not change the value of $b$, then the first expression loops while the second may not (as long as $c$ terminates).

Grading scheme: 1 pt. for “false”. 1 pt. for mentioning nontermination. 3 pts for correct counterexample ($b$ evaluates to true and $c$ does not modify $b$).

3. (5 points) Recall that a program transformation is a function from commands to commands. What does it mean to say that a program transformation is “sound”? (Answer either informally or with a Coq definition.)

Answer:

Informally: A program transformation is sound if $c$ is equivalent to the result of transforming $c$, for every program $c$.

Formally:

Definition $\text{ctrans\_sound}(\text{ctrans} : \text{com} \rightarrow \text{com}) : \text{Prop} :=$

$$\forall (c : \text{com}),\cequiv c (\text{ctrans} c).$$

Grading scheme: -1 for variants on “every command produces the same result as its transformed version” (strictly speaking, they may also fail to produce a result). -2 to -5 for more serious mistakes or nonsense.

4. (7 points) Recall the definition of a valid Hoare triple:

Definition $\text{hoare\_triple}(P:\text{Assertion})(c:\text{com})(Q:\text{Assertion}) : \text{Prop} :=$

$$\forall (st, st'), c/st \rightarrow st' \rightarrow P st \rightarrow Q st'.$$

Indicate whether or not each of the following Hoare triples is valid by writing either “valid” or “invalid.” Where it appears, $a$ is an arbitrary $\text{aexp}$—i.e., you should write “valid” only if the triple is valid for every $a$.

(a) $\{\{\text{True}\}\} X ::= a \{\{X = a\}\}$

Answer: Invalid

(b) $\{\{X = 1\}\}$

$\text{testif (!X === a) then (while BTrue do Y ::= !X) else (Y ::= A0)}$

$\{\{Y = 0\}\}$

Answer: Valid

(c) $\{\{\text{True}\}\}$

$Y ::= A0; Y ::= A1$

$\{\{Y = 1\}\}$

Answer: Valid

(d) $\{\{\text{False}\}\}$

$X ::= A3$

$\{\{X = 0\}\}$

Answer: Valid
5. (9 points) Give the weakest precondition for each of the following commands. (Please use the informal notation for assertions rather than Coq notation—i.e., write \( X = 5 \), not \( \text{fun st} \Rightarrow \text{st} \ X = 5 \).)

(a) \[ \{ \{ ? \} \} X ::= A5 \ \{ \{ X = 5 \} \} \]
   \textbf{Answer: True}
(b) \[ \{ \{ ? \} \} X ::= A0 \ \{ \{ X = 5 \} \} \]
   \textbf{Answer: False}
(c) \[ \{ \{ ? \} \} X ::= !X +++ !Y \ \{ \{ X = 5 \} \} \]
   \textbf{Answer:} \(X + Y = 5\)
(d) \[ \{ \{ ? \} \} \text{ while } A1 \ll = !X \text{ do } (X ::= !X+++A1; \ Y ::= !Y+++A1) \ \{ \{ Y = 5 \} \} \]
   \textbf{Answer:} \(Y - X = 5\)
(e) \[ \{ \{ ? \} \} \text{ while } !X = A0 \text{ do } Y ::= A1 \ \{ \{ Y = 1 \} \} \]
   \textbf{Answer:} \(X=\emptyset \lor Y=1\)
(f) \[ \{ \{ ? \} \}
   \textit{testif} !X = A0
   \text{ then } Y ::= !Z
   \text{ else } Y ::= !W
   \{ \{ Y = 5 \} \} \]
   \textbf{Answer:} \((X=\emptyset \land Z=5) \lor (X<>\emptyset \land W=5)\)

\textbf{Grading scheme: 1 pt for each}

6. (5 points) The notion of weakest precondition has a natural dual: given a precondition and a command, we can ask what is the strongest postcondition of the command with respect to the precondition. Formally, we can define it like this:

Q is the strongest (most difficult to satisfy) assertion that is guaranteed to hold after c if P holds before.

For example, the strongest postcondition of the command \texttt{skip} with respect to the precondition \( Y = 1 \) is \( Y = 1 \). Similarly, the postcondition in...

\[ \{ \{ Y = y \} \}
   \textit{if} !Y = A0 \text{ then } X ::= A0 \text{ else } Y ::= !Y +++ A2
   \{ \{ (Y = y = X = 0) \lor (Y = 2*y \land y <> 0) \} \}\]
...is the strongest one.

Complete each of the following Hoare triples with the strongest postcondition for the given command and precondition.

(a)  {{ Y = 1 }} X ::= !Y +++ A1 {{ ? }}

   Answer: X = 2 \land Y = 1

(b)  {{ True }} X ::= A5 {{ ? }}

   Answer: X = 5

(c)  {{ True }} skip {{ ? }}

   Answer: True

(d)  {{ True }} while BTrue do skip {{ ? }}

   Answer: False

(e)  {{ X = x \land Y = y }}
    while BNot (!X === A0) do (  
       Y ::= !Y +++ A2;  
       X ::= !X --- A1  
    )

   {{ ? }}

   Answer: X = 0 \land Y = y + 2*x

7. (12 points) The following program performs integer division:

    div = 
    Q ::= A0;  
    R ::= ANum x;  
    while (ANum y) <<= !R do (  
       R ::= !R --- (ANum y);  
       Q ::= !Q +++ A1  
    )

If x and y are numbers, running this program will yield a state where Q is the quotient of x by y and R is the remainder. (We assume that program variables Q and R are defined.)

Fill in the blanks in the following to obtain a correct decorated version of the program:

{ 0<y } =>
{ 0=0 \land x=x \land 0<y } =>
Q ::= A0;
{ Q=0 \land x=x \land 0<y } =>
R ::= ANum x;
{ Q=0 \land R=x \land 0<y } =>
{ } =>
while (ANum y <<= !R) do (  
{ } =>

    { } =>
R ::= !R --- (ANum y);
{ } =>
Q ::= !Q +++ A1
{ } =>

3
\[
\begin{align*}
\{ \text{______________________________} \} \Rightarrow \\
\{ x &= Q \times y + R \land R < y \}
\end{align*}
\]

Answer:

\[
\begin{align*}
\{ 0 < y \} \Rightarrow \\
\{ 0 = 0 \land x = x \land 0 < y \}
\end{align*}
\]
\[
\begin{align*}
Q &::= A0; \\
\{ Q = 0 \land x = x \land 0 < y \}
\end{align*}
\]
\[
\begin{align*}
R &::= \text{ANum } x; \\
\{ Q = 0 \land R = x \land 0 < y \} \Rightarrow \\
\{ x = Q \times y + R \}
\end{align*}
\]
\[
\begin{align*}
\text{while } (\text{ANum } y \ll !R) \text{ do } ( \\
\{ x = Q \times y + R \land y \leq R \} \Rightarrow \\
\{ x = (Q+1) \times y + (R-y) \}
\end{align*}
\]
\[
\begin{align*}
R &::= !R \text{ --- (ANum } y); \\
\{ x = (Q+1) \times y + R \}
\end{align*}
\]
\[
\begin{align*}
Q &::= !Q \text{ ++ A1} \\
\{ x = Q \times y + R \}
\end{align*}
\]
\[
\begin{align*}
\{ x = Q \times y + R \land \neg (y \leq R) \} \Rightarrow \\
\{ x = Q \times y + R \land R < y \}
\end{align*}
\]

Grading scheme: -1 for minor errors. -2 for each violation of the rules for forming decorated programs.

8. (4 points) Suppose we change the initial pre-condition in problem 7 from \(0 < y\) to \(\text{True}\) (i.e., we allow \(y\) to be zero). Does the specification now make an incorrect claim — i.e., is the Hoare triple

\[
\{ \text{True} \} \text{ div } \{ \{ x = Q \times y + R \land R < y \} \}
\]

invalid, or is it valid? Briefly explain your answer.

Answer: The specification remains valid: if \(y\) is 0 at the beginning, the program will never terminate and the required condition for validity will hold trivially.

Grading scheme: 2 pts for noting program does not terminate when \(y=0\); 2 pts for stating that Hoare triple is valid for nonterminating program.

9. (6 points) Recall the syntax...

Inductive com : Set :=

| ... \\
| CWhile : bexp \rightarrow com \rightarrow com

...and operational semantics of the while...do... construct:

Inductive ceval : state \rightarrow com \rightarrow state \rightarrow \text{Prop} :=

| ... \\
| CEWhileEnd : forall b1 st c1, 
| beval st b1 = false \rightarrow 
| ceval st (CWhile b1 c1) st \\
| CEWhileLoop : forall st st' st'' b1 c1, 
| beval st b1 = true \rightarrow 
| ceval st c1 st' \rightarrow 
| ceval st' (CWhile b1 c1) st'' \rightarrow 
| ceval st (CWhile b1 c1) st''
Suppose we extend the syntax with one more constructor...

\[ \text{CLoopWhile} : \text{com} \rightarrow \text{bexp} \rightarrow \text{com} \]

...written \text{loop c while b}:

Notation "'loop c 'while' b" := (CLoopWhile c b).

The intended behavior of this construct is almost like that of \text{while...do...} except that the condition is checked at the end of the loop body instead of the beginning (so the body always executes at least once). For example,

\[
\begin{align*}
X &::= A1; \\
\text{loop} &
\begin{align*}
X &::= \lnot X +++ A1 \\
\text{while} &
\begin{align*}
\lnot X &\lll A1
\end{align*}
\end{align*}
\end{align*}
\]

will leave \( X \) with the value 2.

To define the operational semantics of \text{loop...while...} formally, we need to add two more rules to the Inductive declaration of \text{ceval}. Write these rules in the space below.

\[ \begin{align*}
\text{CELoopWhileEnd} : & \forall b1 \text{ st st'} c1, \\
& \text{ceval st c1 st'} \\
& \text{beval st'} b1 = \text{false} \\
& \text{ceval st (CLoopWhile c1 b1) st'}
\end{align*} \]

\[ \begin{align*}
\text{CELoopWhileLoop} : & \forall \text{ st st' st'' b1 c1}, \\
& \text{ceval st c1 st'} \\
& \text{beval st'} b1 = \text{true} \\
& \text{ceval st (CLoopWhile c1 b1) st''} \\
& \text{ceval st (CLoopWhile c1 b1) st''}
\end{align*} \]

\text{Grading scheme: 2 points for rules of the right form with the right conclusion; 2 points for getting the true/false the right way around; 2 points for evaluating the bexp after the command ran instead of before.}

10. (6 points) Having extended the language of commands with \text{loop...while...}, the next thing we want is a Hoare rule for reasoning about programs that use this construct. Recall the rule for \text{while...do...}:

\[
\frac{\{P \land b\} \quad c \quad \{P\}}{\{P\} \quad \text{while } b \quad \text{do } c \quad \{P \land \neg b\}}
\]

Write an analogous rule for \text{loop...while}.

\text{Answer:}

\[
\frac{\{P\} \quad c \quad \{P\}}{\{P\} \quad \text{loop } c \quad \text{while } b \quad \{P \land \neg b\}}
\]

\text{Grading scheme: -3 for minor errors. 0 for major/multiple errors.}

11. (4 points) Recall (from the review session on Monday) the small-step variant of the operational semantics of IMP. The \text{astep} and \text{bstep} relations (not shown here) are small-step reduction relations for \text{aexp}s and \text{bexp}s. The small-step relation for commands is defined as follows:
Inductive cstep : state → com → com → state → Prop :=
| CSAssStep : forall st i a a',
  astep st a a' →
  cstep st (CAss i a) (CAss i a') st
| CSAss : forall st i n,
  cstep st (CAss i (ANum n)) CSkip (extend st i n)
| CSSeqStep : forall st c1 c1' st' c2,
  cstep st c1 c1' st' →
  cstep st (CSeq c1 c2) (CSeq c1' c2) st'
| CSSeqFinish : forall st c2,
  cstep st (CSeq CSkip c2) c2 st
| CSIfTrue : forall st c1 c2,
  cstep st (CIf BTrue c1 c2) c1 st
| CSIfFalse : forall st c1 c2,
  cstep st (CIf BFalse c1 c2) c2 st
| CSIfStep : forall st b b' c1 c2,
  bstep st b b' →
  cstep st (CIf b c1 c2) (CIf b' c1 c2) st
| CSWhile : forall st b c1,
  cstep st (CWhile b c1) (CIf b (CSeq c1 (CWhile b c1)) CSkip) st.

Suppose we extend the syntax of commands with loop...while..., as in the previous two problems. What needs to be added to the definition of cstep?

**Answer 1:**

| CSLoop : forall st b c1,
  cstep st (CLoop c1 b) (CSeq c1 (CWhile b c1) st

**Answer 2:**

| CSLoop : forall st b c1,
  cstep st (CLoop c1 b) (CSeq c1 (CIf b (CLoop b c1) CSkip)) st

**Grading scheme: 1 point for basic syntax, 3 for rule logic.**

12. (12 points) Recall the following definitions from Smallstep.v:

Inductive tm : Set :=
| tm_const : nat → tm
| tm_plus : tm → tm → tm.

Inductive value : tm → Prop :=
| v_const : forall n, value (tm_const n).

Inductive step : tm → tm → Prop :=
| ES_PlusConstConst : forall n1 n2,
  step (tm_plus (tm_const n1) (tm_const n2)) (tm_const (plus n1 n2))
| ES_Plus1 : forall t1 t1' t2,
  (step t1 t1')
  → step (tm_plus t1 t2) (tm_plus t1' t2)
| ES_Plus2 : forall v1 t2 t2',
  (value v1)
  → (step t2 t2')
  → step (tm_plus v1 t2) (tm_plus v1 t2').
In class, we discussed the Progress Theorem:

*Theorem:* If $t$ is a term, then either $t$ is a value or else there exists some term $t'$ such that $t$ steps to $t'$.

Write a careful informal proof of this theorem.

*Answer:*

*Proof:* By induction on $t$.

- Suppose $t = \text{tm}_\text{const} \ n$, then it is a value by $v_\text{const}$.
- If $t = \text{tm}_\text{plus} \ t1 \ t2$ for some $\text{tm} \ t1$ and $t2$, then by the IH $t1$ and $t2$ are either values or can take steps under $\text{step}$.
  - If $t1$ and $t2$ are both values, then $t$ can take a step by $\text{ES}_\text{PlusConstConst}$.
  - If $t1$ is a value and $t2$ can take a step, then so can $t$, by rule $\text{ES}_\text{Plus2}$.
  - Otherwise, $t1$ can take a step. In this case $t$ steps as well, by rule $\text{ES}_\text{Plus1}$.

*Grading scheme:* 1 point for induction, 2 for the base case, 3 for stating the IH in the inductive case. 6 points for the case analysis, reasoning, and clarity of the inductive case.