Solutions
1. (8 points) Put an X in the True or False box for each statement.

(1) For every \( b : \text{bexp} \) and \( c_1, c_2 : \text{com} \), either the command \( \text{IFB} \ b \ \text{THEN} \ c_1 \ \text{ELSE} \ c_2 \ \text{FI} \) is equivalent to \( c_1 \) or it is equivalent to \( c_2 \).

□ True ☒ False

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

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

□ True ☒ False

(Since Imp programs may not terminate, defining evaluation using Fixpoint is awkward.)

(3) For every \( c : \text{com} \) and \( P, Q, R : \text{assertion} \), if \( \{ \{ P \} \ c \ \{ \{ Q \} \} \) is valid, then \( \{ \{ P \land R \} \ c \ \{ \{ Q \lor R \} \} \) is valid.

☒ True □ False

(4) For every \( c : \text{com} \), \( st, st', st'' : \text{state} \), if \( st = [ c ] => st' \) and \( st' = [ c ] => st'' \), then \( st' = st'' \).

□ True ☒ False

(Counterexample: \( c = X ::= X.\!+1 \) and \( b = \text{false} \))

(5) For every \( c : \text{com} \), \( b : \text{bexp} \), and \( P, Q : \text{assertion} \), if the hoare triple \( \{ \{ P \} \ \text{IFB} \ b \ \text{THEN} \ c \ \text{ELSE} \ \text{SKIP} \ \text{FI} \ \{ \{ Q \} \} \) is valid, then \( \{ \{ P \} \ c \ \{ \{ Q \} \} \) is valid.

□ True ☒ False

(Counterexample: Let \( P \) be \( X < 1 \), \( Q \) be \( X < 1 \), and \( c = \text{IFB} \ b \ \text{THEN} \ \text{SKIP} \ \text{ELSE} \ X ::= 1 \ \text{FI} \). Should be \( \{ \{ P \land b \} \ c \ \{ \{ Q \} \} \))

(6) For every \( b : \text{bexp} \) and \( c : \text{com} \), the programs

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

and

\[
\text{WHILE} \ b \ \text{DO}
\]

\[
c ;;
\]

\[
\text{IFB} \ b \ \text{THEN} \ c \ \text{ELSE} \ \text{SKIP} \ \text{END}
\]

END

are equivalent.

☒ True □ False
(7) For every \( b : \text{bexp} \) and \( c_1, c_2 : \text{com} \), the programs

\[
\text{WHILE } b \text{ DO } c_1 \text{ END}
\]

and

\[
\text{WHILE } b \text{ DO } c_1 \text{ END};;
\]

\[
\text{WHILE } b \text{ DO } c_2 \text{ END}
\]

are equivalent.

☑️ True ☐ False

(8) For every \( b_1, b_2 : \text{bexp} \) and \( c : \text{com} \), the programs

\[
\text{WHILE } b_1 \text{ DO}
\]

\[
\quad \text{IFB } b_2 \text{ THEN } c \text{ ELSE SKIP END}
\]

\[
\text{END}
\]

and

\[
\text{WHILE } b_2 \text{ DO}
\]

\[
\quad \text{IFB } b_1 \text{ THEN } c \text{ ELSE SKIP END}
\]

\[
\text{END}
\]

are equivalent.

☐ True ☑️ False

(Counterexample: \( b_1 = \text{true} \) and \( b_2 = \text{false} \))
2. (15 points) Recall that the assertion $P$ appearing in the hoare_while rule (repeated for reference on page 5 in the handout) is called the 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.

(1) \textbf{WHILE} !(X .\leq Z) \textbf{DO} X ::= X .- 1 \textbf{END}

- $\times$ X < 10
- $\Box$ X = 10
- $\times$ X = 0
- $\Box$ X + Z <> 0
- $\times$ Z = 1
- $\times$ Z < 1
- $\times$ True
- $\Box$ None of the above apply

(2) \textbf{WHILE} !(X .= Y) \textbf{DO} Y ::= X;; X ::= Y \textbf{END}

- $\times$ X > 5
- $\Box$ Y > 5
- $\Box$ X + Y > 1
- $\times$ Z = 3 + X
- $\Box$ X * Y <= 1
- $\times$ X = 20
- $\Box$ None of the above apply

(3) \textbf{WHILE} !(X .= Y) \textbf{DO} Y ::= Z;; Z ::= Z .+ 1 \textbf{END}

- $\Box$ Y < 5
- $\Box$ Y > 5
- $\Box$ X = Z
- $\times$ False
- $\Box$ Y = Z + 1
- $\Box$ Y * Z = X
- $\times$ X = 1
- $\Box$ None of the above apply
(4) \[\text{WHILE } X \leq 16 \text{ DO } X := 10 - Y; \quad Y := 1 + Y \text{ END}\]

- Y < 5
- X > 10
- X < 10
- X + Y < 10
- X + Y = 10
- X + Y > 10

\[\text{None of the above apply}\]

(5) \[\text{WHILE } !(X + Y = 0) \text{ DO } X := 10 - Y; \quad Y := 10 - X \text{ END}\]

- Y > X
- X > Y
- X - Y < X + Y
- X + Y > X
- X + Y = X

\[\text{None of the above apply}\]
3. (16 points) For each Hoare triple below, give an invariant for the \texttt{WHILE} loop that will allows us to prove the triple. As usual, \texttt{m} and \texttt{n} are arbitrary values of type \texttt{nat}.

(1) \{\{ X = m \land Y = n \}\}
   
   \begin{verbatim}
   Z ::= 0;;
   WHILE !(X .= 0 \&\& Y .= 0) DO
     IFB !(X .= 0) THEN
       X ::= X .- 1;;
       Z ::= Z .+ 1
     ELSE SKIP FI;;
     IFB !(Y .= 0) THEN
       Y ::= Y .- 1;;
       Z ::= Z .+ 1
     ELSE SKIP FI
   {{ Invariant goes here }}
   END
   {{ Z = m + n }}
   \end{verbatim}

   Invariant = Z + X + Y = m + n

(2) \{\{ X = m \}\}

\begin{verbatim}
Y ::= 0;;
IFB n .= 0 THEN
  Z ::= 1
ELSE
  Z ::= 0;;
  WHILE n .<= X DO
    X ::= X .- n
    Y ::= Y .+ 1
  {{ Invariant goes here }}
END
FI
{{ (Z = 0 \land n * Y + X = m \land X < n) \lor (Z = 1 \land n = 0 \land Y = 0) }}.
\end{verbatim}

   Invariant = Z = 0 \land n * Y + X = m
(3) (The \texttt{min} and \texttt{max} functions are defined on page 10 of the handout.)

\begin{verbatim}
{{ X = m \land Y = n }}
Z ::= 0;;
\textbf{WHILE} Z .\neq 0 \textbf{DO}
  \textbf{IFB} !(X .\leq Y) \textbf{THEN}
    W ::= Y;;
    Y ::= X;;
    X ::= W
  \textbf{ELSE}
    Z ::= 1
  \textbf{FI}
\textbf{END}
{{ X = \texttt{min} m n \land Y = \texttt{max} m n }}

\texttt{Invariant} = (X = \texttt{min} m n \land Y = \texttt{max} m n) \lor (X = m \land Y = n \land Z = 0)
or (Z \neq 0 \rightarrow X = \texttt{min} m n \land Y = \texttt{max} m n) \lor (X = m \land Y = n)
\end{verbatim}

(4) (This command divides an even value stored in $X$ by two.)

\begin{verbatim}
{{ X = 2 \times m }}
Y ::= 0;;
\textbf{WHILE} !(X .\leq Y) \textbf{DO}
  X ::= X .- 1;;
  Y ::= Y .+ 1
\textbf{END}
{{ X = m }}

\texttt{Invariant} = X + Y = 2 \times m \land Y < X + 2
\end{verbatim}
4. (12 points) Translate each of the following informal specifications into a precise Hoare triple by filling in appropriate pre- and postconditions. To reduce clutter, please use informal paper-and-pencil notation when writing assertions (e.g., write $X = 0$ rather than $\text{fun st => st X = 0}$).

For example, if the informal specification is “The command $c$ increases the value of $X$ by 1,” an appropriate pre- and postcondition might be $\{X = m\} \ c \ \{X = m + 1\}$.

1. The command $c$ sets $Z$ to 1 if the initial value of $X$ is between the initial values of $W$ and $Y$ (inclusive); otherwise, it sets $Z$ to 0.

   $$\{\begin{array}{l}
   W = m \land X = n \land Y = p
   \end{array}\} \ c \ \{\begin{array}{l}
   (Z = 1 \land m \leq n \land n \leq p) \lor
   (Z = 0 \land (n < m \lor p < n))
   \end{array}\}$$

2. The command $c$ computes the sum of $X$ and $Y$, diverges if it is 10 or more, and leaves the sum in $Z$ otherwise.

   $$\{\begin{array}{l}
   X = m \land Y = n
   \end{array}\} \ c \ \{\begin{array}{l}
   Z = m + n \land Z < 10
   \end{array}\}$$

3. The command $c$ sets $Z$ to the minimum of the three numbers initially stored in $X$, $Y$, $Z$.

   $$\{\begin{array}{l}
   X = m \land Y = n \land Z = p
   \end{array}\} \ c \ \{\begin{array}{l}
   (Z = m \land Z = n \land Z = p) \lor Z \leq m \land Z \leq n \land Z \leq p
   \end{array}\}$$

   Alternatively, the postcondition can use a function from the appendix:

   $$\{\begin{array}{l}
   Z = \text{min m (min n p)}
   \end{array}\}$$
5. (10 points) Recall the definition of what it means for a term $t$ to be a normal form of a relation $R$:

$$\text{Definition normal_form } \{X : \text{Type}\} (R : \text{relation } X) (t : X) : \text{Prop} := \neg \exists t', R t t'.$$

(1) Give a complete and precise description, in English, of all the Imp arithmetic expressions $a : aexp$ that are normal forms for the $\text{astep}$ relation (i.e., for which the proposition $\text{normal_form } (\text{astep } st) a$ holds for all states $st$).

Answer: For any natural number $n$, $\text{ANum } n$ is in normal form.

(2) Give a complete and precise description of all the Imp commands $c$ and states $st$ such that the pair $(c, st)$ is a normal form for the $\text{cstep}$ relation.

Answer: The only Imp program in normal form is $\text{SKIP}$; it can be paired with any state.

(3) Define informally what it means for a step relation to be normalizing.

Answer: A step relation is normalizing if, from any starting state, it reaches a normal form in a finite number of steps.

(4) Give an informal definition of strong progress.

Answer: A relation has the strong progress property if every $t$ either is a value or can take a step.

(5) Does strong progress imply termination?

☐ Yes ☒ No
6. **[Standard Track Only]** (5 points) Recall that

- A Hoare triple $\{P\} c \{Q\}$ is valid if, whenever $st$ is a state satisfying $P$ and $st = [c] \Rightarrow st'$, the final state $st'$ satisfies $Q$, and

- $\{P\} c \{Q\}$ is invalid if it is not valid.

Consider the following additional properties of Hoare triples:

- $\{P\} c \{Q\}$ is nontrivial if there is some pair of starting and ending states $st$ and $st'$ such that $st$ satisfies $P$ and $st = [c] \Rightarrow st'$ (i.e., if $c$ sometimes terminates when started on a state satisfying $P$). It is trivial if $c$ never terminates when started on a state satisfying $P$.

- $\{P\} c \{Q\}$ is quite valid if it is nontrivial and valid.

- $\{P\} c \{Q\}$ is quite invalid if $\{P\} c \{\sim Q\}$ is quite valid.

- $\{P\} c \{Q\}$ is less than valid if it is neither valid nor quite invalid.

To get warmed up, let’s check how these terms are related. (Please mark the appropriate box.)

1. If $\{P\} c \{Q\}$ is less than valid, then it is nontrivial.
   - [ ] True
   - [ ] False

2. If $\{P\} c \{Q\}$ is quite invalid, then it is nontrivial.
   - [ ] True
   - [ ] False

3. If $\{P\} c \{Q\}$ is less than valid, then it is invalid.
   - [ ] True
   - [ ] False

4. If $\{P\} c \{Q\}$ is nontrivial, then it is either quite valid or quite invalid.
   - [ ] True
   - [ ] False

5. If $\{P\} c \{Q\}$ is trivial, then it is valid.
   - [ ] True
   - [ ] False
7. [Standard Track Only] (14 points) (Continuing the previous problem) Now, for each of the following Hoare triples, choose the property that correctly describes it:

(1) \[
\{ X = 0 \} \\
\text{WHILE !(X .\neq 0) DO SKIP END} \\
\{ False \}
\]
- trivial
- quite valid
- less than valid
- quite invalid

(2) \[
\{ True \} \\
\text{WHILE !(X .\leq 1) DO X := X .- 1 END} \\
\{ X = 0 \}
\]
- trivial
- quite valid
- less than valid
- quite invalid

(3) \[
\{ 1 \leq Z \} \\
\text{IFB Z .\leq 2 THEN Z := Z .+ 1 ELSE SKIP FI} \\
\{ 2 \leq Z \}
\]
- trivial
- quite valid
- less than valid
- quite invalid

(4) \[
\{ X < Z \} \\
\text{WHILE !(Z .\leq X) DO X := X .+ 1;; Z := Z .- 1 END} \\
\{ X = Z \}
\]
- trivial
- quite valid
- less than valid
- quite invalid

(5) \[
\{ X \leq 1 \land Y = 1 \} \\
\text{WHILE X .\leq Y DO X := X .* Y END} \\
\{ Y < X \}
\]
- trivial
- quite valid
- less than valid
- quite invalid

(6) \[
\{ X \leq 1 \} \\
\text{WHILE X .\neq 1 DO SKIP END} \\
\{ X \leq 1 \}
\]
- trivial
- quite valid
- less than valid
- quite invalid

(7) \[
\{ True \} \\
\text{WHILE X .\neq Y DO X := X .+ 1;; Y := Y .+ 1 END} \\
\{ X = Y \}
\]
- trivial
- quite valid
- less than valid
- quite invalid
8. **[Advanced Track Only]** (19 points) Below is the start of a detailed informal proof that the program

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

is equivalent to this one:

\[
\text{IFB } X .<= Y \\
\text{THEN } X ::= Y \\
\text{ELSE WHILE true DO SKIP END FI}
\]

Fill in the proof for claim (1) below. You may freely use the lemmas about updating total maps from Maps.v (repeated on page 1 of the handout, for reference) without explicitly saying you are doing so. Use the next page if you need more space.

**Proof:** Let \( c_1 \) be the first program and \( c_2 \) the second. The definition of equivalence says that, given states \( st \) and \( st' \), we must show that (1) \( st = \llbracket c_1 \rrbracket \Rightarrow st' \) implies \( st = \llbracket c_2 \rrbracket \Rightarrow st' \) and (2) \( st = \llbracket c_2 \rrbracket \Rightarrow st' \) implies \( st = \llbracket c_1 \rrbracket \Rightarrow st' \).

Fill in just the proof of claim (1)...

**Solution, variant 1:**

We first show, by induction on a derivation of \( st = \llbracket c_1 \rrbracket \Rightarrow st' \), that \( st \ X <= st \ Y \) and \( st' = (X !-> st \ Y; st) \). The last rule in this derivation must be either E_WhileFalse or E_WhileTrue.

- **E_WhileFalse:** The premises of the rule give us \( st' = st \) and \( \text{beval}(st \ !(X . = Y)) = \text{false} \), i.e., \( st \ X = st \ Y \). It follows by the definition of \( <= \) that \( st \ X <= st \ Y \) and from the properties of map update that \( st' = (X !-> st \ Y; st) \).

- **E_WhileTrue:** The premises of the rule give us \( st = \llbracket X := X .+ 1 \rrbracket \Rightarrow st'' \) and \( st'' = \llbracket c_1 \rrbracket \Rightarrow st' \) (and \( \text{beval}(st \ !(X . = Y)) = \text{false} \), though we do not use this). The induction hypothesis gives us \( st'' \ X <= st'' \ Y \) and \( st' = (X !-> st'' \ Y; st'') \).

Now, the final rule in the derivation of \( st = \llbracket X := X .+ 1 \rrbracket \Rightarrow st'' \) must be E_Ass, so \( st'' = (X !-> st \ X + 1; st) \), hence \( st \ X < st'' \ X \) and \( st \ Y = st'' \ Y \), and so \( st \ X <= st \ Y \). Moreover, \( st \) and \( st'' \) differ only at \( X \), so \( (X !-> st'' \ Y; st'') = (X !-> st \ Y; st) \) by the properties of map update. Thus, \( st' = (X !-> st \ Y; st) \) as required.

To finish, observe that, by E_Ass, \( st = \llbracket X := Y \rrbracket \Rightarrow (X !-> Y; st) \). Using the fact that \( st \ X <= st \ Y \), rule E_IfTrue yields \( st = \llbracket c_2 \rrbracket \Rightarrow (X !-> Y; st) = st' \).

**Variant 2:**

We show, by induction on a derivation of \( st = \llbracket c_1 \rrbracket \Rightarrow st' \), that \( st = \llbracket c_2 \rrbracket \Rightarrow st' \). Since \( c_1 \) is a WHILE command, there are two cases to consider:

- The final rule is E_WhileFalse. Then \( \text{beval}(st \ !(X .= Y)) = \text{false} \), so \( st \ X = st \ Y \), and \( st = st' \). It follows that \( st \ X <= st \ Y \); from this, E_IfTrue and E_Ass yield \( st = \llbracket c_2 \rrbracket \Rightarrow (X !-> st \ Y; st) \). Finally, \( (X !-> st \ Y; st) = (X !-> st \ X; st) = st = st' \).
The final rule is E\_WhileTrue. Then beval st ! (X = Y) = true (so st X <> st Y) and we are given

Now, because c2 is an IFB command, st'' = [c2] => st' must have been derived in one of two ways:

- The final rule is E\_IfTrue. Then st'' X <= st'' Y, and st'' = [X := Y] => st'. Therefore, st''' = (X !-> Y; st') = (X !-> Y; st). But we know that st X = st X + 1, and st Y = st Y, so by E\_IfTrue, st = [c2] => st'.

- The final rule is E\_IfFalse. Then st'' = [WHILE true DO SKIP END] => st'. But this is a contradiction, since we know that this program diverges on all inputs.
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 =\([\text{SKIP}]\) => 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 =\([\text{IFB } b \text{ THEN } c1 \text{ ELSE } c2 \text{ FI}]\) => st’ 
| E_IfFalse : forall st st’ b c1 c2, 
  beval st b = false ->  
  st =\([c2]\) => st’ ->  
  st =\([\text{IFB } b \text{ THEN } c1 \text{ ELSE } c2 \text{ FI}]\) => st’ 
| E_WhileFalse : forall b st c, 
  beval st b = false ->  
  st =\([\text{WHILE } b \text{ DO } c \text{ END}]\) => st 
| E_WhileTrue : forall st st’ st’’ b c, 
  beval st b = true ->  
  st =\([c]\) => st’ ->  
  st =\([\text{WHILE } b \text{ DO } c \text{ END}]\) => st’’ ->  
  st =\([\text{WHILE } b \text{ DO } c \text{ END}]\) => 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 triples

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

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

Hoare logic rules

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

\[ \{ \text{assn_sub } x \overset{a}{\Rightarrow} Q \} x := a \{ Q \} \] (hoare_asgn)

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

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

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

\[ \{ P \land b \} c \{ P \} \]
\[ \{ P \} \text{WHILE} b \text{ DO } c \text{ END } \{ P \land \lnot b \} \] (hoare_while)

\[ \begin{align*}
\{ P' \} c \{ Q \} \\
P \rightarrow P'
\end{align*} \] (hoare_consequence_pre)

\[ \begin{align*}
\{ P \} c \{ Q' \} \\
Q' \rightarrow Q
\end{align*} \] (hoare_consequence_post)
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'.

Lemmas for the multi-step 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.
Small Step Semantics

Small-step relation for arithmetic expressions

Inductive astep : state -> aexp -> aexp -> Prop :=
| AS_Id : forall st i,
  AId i / st -->a ANum (st i)
| 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_Plus : forall st n1 n2,
  APlus (ANum n1) (ANum n2) / st -->a ANum (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_Minus : forall st n1 n2,
  (AMinus (ANum n1) (ANum n2)) / st -->a (ANum (minus 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')
| AS_Mult : forall st n1 n2,
  (AMult (ANum n1) (ANum n2)) / st -->a (ANum (mult n1 n2))

where " t '/\' st '->a' t' " := (astep st t t').
Small-step relation for boolean expressions

Inductive bstep : state -> bexp -> bexp -> Prop :=
| BS_Eq1 : forall st a1 a1' a2, a1 / st -->a a1' -> (BEq a1 a2) / st -->b (BEq a1' a2)
| BS_Eq2 : forall st v1 a2 a2', aval v1 -> a2 / st -->a a2' -> (BEq v1 a2) / st -->b (BEq v1 a2')
| BS_Eq : forall st n1 n2, (BEq (ANum n1) (ANum n2)) / st -->b
  if (n1 =? n2) then BTrue else BFalse
| BS_LtEq1 : forall st a1 a1' a2, a1 / st -->a a1' -> (BLe a1 a2) / st -->b (BLe a1' a2)
| BS_LtEq2 : forall st v1 a2 a2', aval v1 -> a2 / st -->a a2' -> (BLe v1 a2) / st -->b (BLe v1 a2')
| BS_LtEq : forall st n1 n2, (BLe (ANum n1) (ANum n2)) / st -->b
  if (n1 <=? n2) then BTrue else BFalse
| BS_NotStep : forall st b1 b1', b1 / st -->b b1' -> (BNot b1) / st -->b (BNot b1')
| BS_NotTrue : forall st, (BNot BTrue) / st -->b BFalse
| BS_NotFalse : forall st, (BNot BFalse) / st -->b BTrue
| BS_AndTrueStep : forall st b2 b2', b2 / st -->b b2' -> (BAnd BTrue b2) / st -->b (BAnd BTrue b2')
| BS_AndStep : forall st b1 b1' b2, b1 / st -->b b1' -> (BAnd b1 b2) / st -->b (BAnd b1' b2)
| BS_AndTrueTrue : forall st, (BAnd BTrue BTrue) / st -->b BTrue
| BS_AndTrueFalse : forall st, (BAnd BTrue BFalse) / st -->b BFalse
| BS_AndFalse : forall st b2, (BAnd BFalse b2) / st -->b BFalse

where " t '/' st '-->b' t' " := (bstep st t t').
Small-step relation for Imp commands

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 / (i !-> n ; st)
  | 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_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_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_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')).
Simple expression language

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

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

Big-step evaluation of tm

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).

Small-step relation for tm

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').

Theorem strong_progress : forall t,
  value t \/ (exists t', t --> t').

Minimum and maximum functions

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

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