Solutions
(11 points) Translate each of the following informal specifications into a precise formal specification using a Hoare triple. For example, if the informal specification is “The command $c$ increases the value of $X$ by 1,” an appropriate formal specification might be $\forall m, \{ \{ X = m \} \} c \{ \{ X = m + 1 \} \}$. To reduce clutter, please use informal paper-and-pencil notation when writing assertions (e.g., write $X = 0$ rather than $\text{fun st => st X = 0}$).

If you use Coq variables in your pre- and post-conditions (like $m$ above), make sure that you write an explicit quantifier before the precondition.

1.1 The command $c$ swaps the values of $X$ and $Y$.

$$\forall n m, \quad \{ \{ X = n \land Y = m \} \} c \quad \{ \{ Y = m \land X = n \} \}$$

1.2 The command $c$ sets $Z$ to be the maximum of $X$ and $Y$.

$$\forall n m, \quad \{ \{ X = m \land Y = n \} \} c \quad \{ \{ (m > n \land Z = m) \lor (m \leq n \land Z = n) \} \}$$

1.3 When $Y$ is non-zero, $c$ computes $X / Y$ and stores the result in $Z$.

$$\forall x y, \quad \{ \{ Y = y \land X = x \land Y \neq 0 \} \} c \quad \{ \{ Z = x / y \} \}$$

1.4 The command $c$ diverges if the initial value of $X$ is strictly smaller than the initial value of $Y$.

$$\forall n m, \quad \{ \{ X = n \land Y = m \} \} c \quad \{ \{ n \geq m \} \}$$
2. (12 points) For each of the following pairs of programs, mark the box indicating whether or not the programs are equivalent according to $cequiv$. If the programs are not equivalent, give a counterexample—i.e., a starting state on which they behave differently.

Note that some of these definitions are actually “templates” for commands (e.g., $?\text{?}$ is implicitly parameterized by a natural number $n$ and $?\text{??}$ by an arbitrary boolean expression $b$). For these you should make sure that the programs are equivalent for any choice of the parameter.

2.1

<table>
<thead>
<tr>
<th>□ Equivalent</th>
<th>☒ Not Equivalent</th>
</tr>
</thead>
</table>

**Counterexample (if “Not Equivalent”):** $st=(X!->1)$

2.2

| SKIP | T ::= X;; X ::= Y;; Y ::= T;; |
|      | T ::= X;; X ::= Y;; Y ::= T |

<table>
<thead>
<tr>
<th>□ Equivalent</th>
<th>☒ Not Equivalent</th>
</tr>
</thead>
</table>

**Counterexample (if “Not Equivalent”):** $st=(T!->1)$

2.3

| WHILE b DO | Y ::= X;; |
|            | Z ::= Y * Z |
|            | END |
| Y ::= X;; | WHILE b DO |
|            | Z ::= Y * Z |
|            | END |

<table>
<thead>
<tr>
<th>□ Equivalent</th>
<th>☒ Not Equivalent</th>
</tr>
</thead>
</table>

**Counterexample (if “Not Equivalent”):** $st=(Y!->1; X!->0)$, if $b$ evaluates to false

2.4

| WHILE $\neg(X = Y)$ DO | X ::= X - 1;; |
| X ::= X + 1 |
| END |
| X ::= X - 1;; | WHILE $\neg(X = Y)$ DO |
| X ::= X + 1 |
| END |

<table>
<thead>
<tr>
<th>□ Equivalent</th>
<th>☒ Not Equivalent</th>
</tr>
</thead>
</table>

**Counterexample (if “Not Equivalent”):** $st=(X!->1; Y!->0)$
In order to verify an Imp program that contains a loop—either with a direct proof using the Hoare Logic rules (page ?? in the appendix) or by decorating it with assertions (page ?? in the appendix)—we need to find a loop invariant that also serves as a valid pre- and post-condition for the loop. That is, we need to find an assertion that satisfies three conditions:

(a) It must be weak enough to be implied by the loop’s precondition.

(b) It must be strong enough to imply the program’s postcondition (given that the loop guard evaluates to false).

(c) It must be preserved by each iteration of the loop (given that the loop guard evaluates to true).

For example, the assertion True is not a useful invariant for the following decorated program because it fails to imply the needed postcondition:

(1) \{X = m \land Y = n\} \implies (a) OK - (1) implies (2)
(2) \{True\}
(3) \{True \land X \neq 0\} \implies (c) OK - (3) implies (4)
(4) \{True\}
(5) Y ::= Y - 1
(6) \{True\}
(7) \{True \land \neg(X \neq 0)\} \implies (b) WRONG! - (7) doesn’t imply (8)
(8) \{Y = n - m\}

For each of the proposed invariants for the decorated WHILE loop at the top of the next page, mark all of the correct implications—e.g., for the example loop on this page and the proposed invariant True, you would mark (a) and (c) but not (b).
Decorated loop:

\[
\begin{align*}
\{ \{ X = n \land Y = 0 \} \} & \Rightarrow (a) \\
\{ \{ \text{invariant} \} \} & \\
\text{WHILE } \neg(X = 0) \text{ DO} & \\
\{ \{ \text{invariant} \land X \neq 0 \} \} & \Rightarrow (c) \\
\{ \{ \ldots \} \} & \\
X & := X - 1; \\
Y & := Y + 1 \\
\{ \{ \text{invariant} \} \} & \\
\text{END} & \\
\{ \{ \text{invariant} \land (X = 0) \} \} & \Rightarrow (b) \\
\{ \{ Y = n \} \} &
\end{align*}
\]

Possible invariants:

3.1 \( X \leq n \)

- (a) Precondition implies invariant
- (b) Invariant and negated guard imply postcondition
- (c) Invariant is preserved by the loop body

3.2 \( Y - X > 0 \)

- (a) Precondition implies invariant
- (b) Invariant and negated guard imply postcondition
- (c) Invariant is preserved by the loop body

3.3 \( Y > 0 \)

- (a) Precondition implies invariant
- (b) Invariant and negated guard imply postcondition
- (c) Invariant is preserved by the loop body

3.4 \( X + Y = n \land X \leq n \)

- (a) Precondition implies invariant
- (b) Invariant and negated guard imply postcondition
- (c) Invariant is preserved by the loop body

3.5 False

- (a) Precondition implies invariant
- (b) Invariant and negated guard imply postcondition
- (c) Invariant is preserved by the loop body
The Imp program below calculates the sum of the natural numbers up to \( n \). Add appropriate annotations to the 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 \textit{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 \( \rightarrow \rightarrow \).

The implication steps in your decorations may rely (silently) on all the usual rules of natural-number arithmetic.

The rules of Hoare logic and the rules for well-formed decorated programs can be found on pages 3 and ?? of the appendix.

\[
\begin{align*}
\{ & \ X = n \ \} \rightarrow > > \\
\{ & \ 2 \ast 0 = 0 \ast (1 + 0) \sqcap (X = n) \} \\
\ Y := 0 \\
\{ & \ 2 \ast Y = 0 \ast (1 + 0) \sqcap (X = n) \} ; ; \\
\ Z := 0 \\
\{ & \ 2 \ast Y = Z \ast (1 + Z) \sqcap (X = n) \} ; ; \\
\text{WHILE } \neg (Z = X) \text{ DO} \\
\{ & \ (2 \ast Y = Z \ast (1 + Z) \sqcap (X = n)) \sqcap (Z < X) \} \rightarrow > > \\
\{ & \ 2 \ast (Y + (1 + Z)) = (1 + Z) \ast (1 + (1 + Z)) \sqcap (X = n) \} \\
\ Z := 1 + Z \\
\{ & \ 2 \ast (Y + Z) = Z \ast (1 + Z) \sqcap (X = n) \} ; ; \\
\ Y := Y + Z \\
\{ & \ 2 \ast Y = Z \ast (1 + Z) \sqcap (X = n) \} \\
\text{END} \\
\{ & \ (2 \ast Y = Z \ast (1 + Z) \sqcap (X = n)) \sqcap (Z = X) \} \rightarrow > > \\
\{ & \ 2 \ast Y = n \ast (1 + n) \} 
\end{align*}
\]
The two following programs compute the sum of natural numbers from 0 to \( n \) in different ways:

**Definition sum_loop : com :=**

\[
\text{WHILE } \neg (C = 0) \text{ DO}
\]

\[
T ::= T + C ; ; \\
C ::= C - 1
\]

**END.**

**Definition sum_closed : com :=**

\[
T ::= T + (C \times (C + 1)) / 2 ; ; \\
C ::= 0
\]

The first program (**sum_loop**) calculates the sum incrementally, adding \( n \) (the initial value of \( C \)) to \( T \), then adding \( n-1 \) to \( T \), and so on. The second (**sum_closed**) just statically adds the closed form of the sum from 0 to \( n \) to \( T \) (and sets \( C \) to 0).

Note that, in both cases, we *add* the sum to the initial value of \( T \) rather than setting \( T \) to the sum. This makes the reasoning below a bit simpler. Also, note that we have technically added division to the IMP language here. You can assume that it behaves like normal integer division, where the result is rounded down (details like rounding and how division by zero behaves should not matter here).

On the next page, write a *careful, informal* proof of (one direction of) the equivalence between these programs. Be sure to spell out any induction hypotheses explicitly.

The following lemma may help simplify some calculation in your proof:

**Lemma sn_sum :**

\[
\text{forall } n, S n + ((n \times (n + 1)) / 2) = (S n \times (S n + 1)) / 2.
\]

Additionally, to keep the proof smaller, if you have a command with multiple assignments sequenced together like so...

\[
\text{st } = [ X ::= 2 ; ; Y ::= 2 + X ; ; c ] => st'
\]

...you may derive in one step that:

\[
(X !-> 2 ; Y !-> 4 ; st) = [ c ] => st'
\]

Normally you would arrive at this conclusion by multiple inversions of the first evaluation relation, but this level of detail is not necessary. It is helpful to phrase the theorem in a slightly odd way, so that we can prove it by induction on \( n \) instead of induction on ceval derivations (this would also work, but the proof would be a bit heavier):

**Theorem sum_loop_implies_sum_closed :**

\[
\text{forall } (n : \text{nat}), \\
\text{forall } (st \ st' : \text{state}), \\
\text{st C = n } => \\
\text{st = [sum_loop] => st' } => \\
\text{st = [sum_closed] => st' .}
\]

**Proof:** By induction on \( n \).

[BCP: Still needs some polishing: I see several bits of displayed Coq that need to be indented, a comma at the beginning of a line, and at least two incomplete sentences]
the loop guard of `sum_loop` is false. Thus the loop is never entered, and we're left in a final state where \( T \) and \( C \) (and \( C=n=0 \)) are unchanged. Similarly for `sum_closed`, \( C = 0 \), and \( T = T + 0 = T \). Both programs don’t change the state.

In the inductive case, where \( n = S \ n' \), we wish to prove

```plaintext
forall st st',
st C = S n' ->
st = [sum_loop] => st' ->
st = [sum_closed] => st'
```

given the inductive hypothesis:

```plaintext
forall st st',
st C = n' ->
st = [sum_loop] => st' ->
st = [sum_closed] => st'
```

Since \( st C = S n' \), only the `E_WhileTrue` rule of `ceval` could have been used for constructing \( st = [sum_loop] => st' \). The hypotheses of `E_WhileTrue` are:

- \( st = [ T ::= T + C;; C ::= C - 1 ] => st'0 \); and
- \( st'0 = [\ WHILE b DO c END ] => st', i.e., st'0 = [ sum_loop ] => st' \).

From \( st = [ T ::= T + C;; C ::= C - 1 ] => st'0 \) we can conclude that

\[
\begin{align*}
  st'0 &= (T !-> T + (S n'); C !-> n'; st) \\

t &\rightarrow n' \text{ and sum loops}
\end{align*}
\]

and thus, by substituting into \( st'0 = [ sum_loop ] => st' \), we also know:

\[
\begin{align*}
  (T !-> T + (S n')); C !-> n'; st) &= [ sum_loop ] => st' \\
\end{align*}
\]

Applying the inductive hypothesis to this yields

\[
\begin{align*}
  (T !-> T + (S n')); C !-> n'; st) &= [ sum_closed ] => st' \\
\end{align*}
\]

and thus we can conclude that \( st' = (T !-> T + (S n')) + (n' * (n' + 1)) / 2; C !-> 0; st) \). Applying `sn_sum` in \( st' \) we can conclude that:

\[
\begin{align*}
  st' &= (T !-> T + (S n' * (S n' + 1)) / 2; C !-> 0; st) \\
\end{align*}
\]

Finally, we need to show \( st = [sum_closed] => st' \), which from our assumptions can be rewritten to:

\[
\begin{align*}
  st = [sum_closed] => (T !-> T + (S n' * (S n' + 1)) / 2; C !-> 0; st) \\
\end{align*}
\]

This follows by applying `E_Seq` and `E_Ass`.  

7
The version of Hoare Logic that we’ve studied in class focuses on *partial correctness*—i.e., its specifications (“Hoare triples”) have the form “if we start command $c$ in a state satisfying precondition $P$, and if $c$ terminates in a state $st'$, then $st'$ satisfies postcondition $Q$.”

There is another variant of Hoare Logic, called *total-correctness* Hoare Logic, whose triples make a stronger claim about termination: “if we start command $c$ in a state satisfying $P$, then $c$ is guaranteed to terminate in some state $st'$ that satisfies $Q$.” This stronger form of Hoare triple is defined as follows:

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

We write total-correctness Hoare triples with $[[\ldots]]$ brackets around the pre- and post-conditions, to distinguish them from our familiar partial-correctness triples.

\[
\text{Notation } "[[ P ]] c [[ Q ]]" := (\text{tc_hoare_triple P c Q})
\]
On the left below are a number of ordinary (partial correctness) Hoare triples; on the right are analogous total-correctness triples. Indicate whether each triple is valid or invalid in the appropriate sense by checking the corresponding box. Note that the commands and assertions are the same between the left and right parts of each subproblem—only the shape of the brackets around the assertions (i.e., whether we are talking about partial or total correctness) is different.

### Partial correctness:

#### {{ True }}

```plaintext
WHILE 1 <= X DO
    X ::= X - 1
END
{{ X = 0 }}
```

- □ Valid
- □ Invalid

### Total correctness:

#### [[ True ]]

```plaintext
WHILE 1 <= X DO
    X ::= X - 1
END
[[ X = 0 ]]
```

- □ Valid
- □ Invalid

### Partial correctness:

#### {{ True }}

```plaintext
WHILE 1 <= X DO
    X ::= X + 1
END
{{ True }}
```

- □ Valid
- □ Invalid

### Total correctness:

#### [[ True ]]

```plaintext
WHILE 1 <= X DO
    X ::= X + 1
END
[[ True ]]
```

- □ Valid
- □ Invalid

### Partial correctness:

#### {{ X = 0 }}

```plaintext
WHILE 1 <= X DO
    X ::= X + 1
END
{{ True }}
```

- □ Valid
- □ Invalid

### Total correctness:

#### [[ X = 0 ]]

```plaintext
TEST X = 5 THEN
    WHILE true DO SKIP END
ELSE
    SKIP
FI
[[ X = 3 ]]
```

- □ Valid
- □ Invalid

### Partial correctness:

#### {{ X = 5 }}

```plaintext
TEST X = 5 THEN
    WHILE true DO SKIP END
ELSE
    SKIP
FI
{{ X = 3 }}
```

- □ Valid
- □ Invalid

### Total correctness:

#### [[ X = 5 ]]

```plaintext
TEST X = 5 THEN
    WHILE true DO SKIP END
ELSE
    SKIP
FI
[[ X = 3 ]]
```

- □ Valid
- □ Invalid
6.2 Are the following propositions true? If so, check Yes. If not, check No and give a counterexample.

forall (c : com), \[[\text{False}]\] c \[[\text{True}]\]

☑ Yes    □ No

Counterexample (if appropriate):
forall (c : com), \[[\text{True}]\] c \[[\text{True}]\]

□ Yes ☑ No

Counterexample (if appropriate): c = WHILE true DO SKIP END
forall P, \[[P]\] WHILE true DO SKIP END \[[P]\]

□ Yes ☑ No

Counterexample (if appropriate): P = True
forall P, \[[P]\] WHILE false DO SKIP END \[[P]\].

☑ Yes    □ No

Counterexample (if appropriate):
(10 points) Recall the definition of what it means for a term \( t \) to be a \textit{normal form} of a relation \( R \):

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

7.1 Give a complete and precise description, in English, of all the Imp arithmetic expressions \( a : aexp \) that are normal forms for the small-step reduction relation on arithmetic expressions (\textit{astep}, see page ?? in the appendix)—i.e., for which the proposition \text{normal_form} (\text{astep} st) a holds for all states \( st \).

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

7.2 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 \textit{cstep} relation (page ?? in the appendix).

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

7.3 Define informally what it means for a step relation to be \textit{normalizing}.

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

7.4 Give an informal definition of strong progress.

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

7.5 Does strong progress imply termination?

\begin{itemize}
\item \( \square \) Yes
\item \( \Box \) No
\end{itemize}
Use this space for scratch work that you don't want graded. If you write something here that you do want graded, make sure there is a very clear pointer from the earlier page where you ran out of space.
For Reference

Imp

Arithmetic expressions (extended with division) and boolean expressions:

\[
\text{Fixpoint } \text{aeval} \ (\text{st} : \text{state}) \ (a : \text{aexp}) : \text{nat} :=
\begin{align*}
\text{match } a \text{ with} \\
| \text{ANum} \ n & \Rightarrow n \\
| \text{AId} \ x & \Rightarrow \text{st} \ x \\
| \text{APlus} \ a1 \ a2 & \Rightarrow (\text{aeval} \ \text{st} \ a1) + (\text{aeval} \ \text{st} \ a2) \\
| \text{AMinus} \ a1 \ a2 & \Rightarrow (\text{aeval} \ \text{st} \ a1) - (\text{aeval} \ \text{st} \ a2) \\
| \text{AMult} \ a1 \ a2 & \Rightarrow (\text{aeval} \ \text{st} \ a1) \times (\text{aeval} \ \text{st} \ a2) \\
| \text{ADiv} \ a1 \ a2 & \Rightarrow (\text{aeval} \ \text{st} \ a1) / (\text{aeval} \ \text{st} \ a2)
\end{align*}
\]

\[
\text{Fixpoint } \text{beval} \ (\text{st} : \text{state}) \ (b : \text{bexp}) : \text{bool} :=
\begin{align*}
\text{match } b \text{ with} \\
| \text{BTrue} & \Rightarrow \text{true} \\
| \text{BFalse} & \Rightarrow \text{false} \\
| \text{BEq} \ a1 \ a2 & \Rightarrow (\text{aeval} \ \text{st} \ a1) =? (\text{aeval} \ \text{st} \ a2) \\
| \text{BLe} \ a1 \ a2 & \Rightarrow (\text{aeval} \ \text{st} \ a1) <=? (\text{aeval} \ \text{st} \ a2) \\
| \text{BNot} \ b1 & \Rightarrow \text{negb} (\text{beval} \ \text{st} \ b1) \\
| \text{BAnd} \ b1 \ b2 & \Rightarrow \text{andb} (\text{beval} \ \text{st} \ b1) (\text{beval} \ \text{st} \ b2)
\end{align*}
\]

Commands:

\[
\begin{align*}
\text{----------------- (E_Skip)} \\
& \text{st } =\text{[ SKIP ]} \Rightarrow \text{st} \\
\text{----------------- (E_Ass)} \\
& \text{aeval} \ \text{st} \ a1 = n \\
& \text{----------------- (E_Seq)} \\
& \text{st } =\text{[ x := a1 ]} \Rightarrow (\text{x } !-> \ n ; \text{st}) \\
& \text{------------- (E_Ass)} \\
& \text{st } =\text{[ c1 ]} \Rightarrow \text{st}' \\
& \text{st}' =\text{[ c2 ]} \Rightarrow \text{st}'' \\
& \text{------------- (E_Seq)} \\
& \text{st } =\text{[ c1; c2 ]} \Rightarrow \text{st}'' \\
\text{----------------- (E_IfTrue)} \\
& \text{beval} \ \text{st} \ b1 = \text{true} \\
& \text{st } =\text{[ c1 ]} \Rightarrow \text{st}' \\
& \text{----------------- (E_IfFalse)} \\
& \text{beval} \ \text{st} \ b1 = \text{false} \\
& \text{st } =\text{[ c2 ]} \Rightarrow \text{st}'
\end{align*}
\]
beval st b = false
----------------------------- (E_WhileFalse)
st =[ WHILE b DO c END ]=> st

beval st b = true
st =[ c ]=> st'
st' =[ WHILE b DO c END ]=> st''
----------------------------- (E_WhileTrue)
st =[ WHILE b DO c END ]=> st''

Equivalence

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

Hoare triples

Definition Assertion := state -> Prop.

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

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

Definition hoare_triple
  (P : Assertion) (c : com) (Q : Assertion) : Prop :=
  forall st st',
  st =[ c ]=> st' ->
  P st ->
  Q st'.

Notation "{{ P }} c {{ Q }}" := (hoare_triple P c Q)
Hoare logic

\[
\begin{align*}
\text{\texttt{hoare\_asgn}} & \quad \{Q \ [X \rightarrow a]\} \ X := a \ \{Q\} \\
\text{\texttt{hoare\_skip}} & \quad \{P\} \ \text{SKIP} \ \{P\} \\
\text{\texttt{hoare\_seq}} & \quad \{P\} \ c_1 \ \{Q\} \quad \{Q\} \ c_2 \ \{R\} \\
\text{\texttt{hoare\_if}} & \quad \{P\} \ \text{TEST} \ b \ \text{THEN} \ c_1 \ \text{ELSE} \ c_2 \ \text{FI} \ \{Q\} \\
\text{\texttt{hoare\_while}} & \quad \{P\} \ \text{WHILE} \ b \ \text{DO} \ \ c \ \text{END} \ \{P \land \neg b\} \\
\text{\texttt{hoare\_consequence}} & \quad \{P\} \ c \ \{Q\} \\
\end{align*}
\]
Decorated programs

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

\[ \{\{ P \}\} \text{SKIP} \{\{ P \}\} \]

(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 \( R \)):

\[ \{\{ P \}\} \quad c_1 \quad ; \quad \{\{ Q \}\} \quad c_2 \quad \{\{ R \}\} \]

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

\[ \{\{ P \}[X \mapsto a]\} \quad X := a \quad \{\{ P \}\} \]

(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 \( Q \)):

\[ \begin{align*}
\{\{ P \}\} \\
\text{IFB} \quad b \quad \text{THEN} \\
\{\{ P \land b \}\} \quad c_1 \quad \{\{ Q \}\} \\
\text{ELSE} \\
\{\{ P \land \neg b \}\} \quad c_2 \quad \{\{ Q \}\} \\
\text{FI} \\
\{\{ Q \}\}
\end{align*} \]

(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 \( P \):

\[ \begin{align*}
\{\{ P \}\} \\
\text{WHILE} \quad b \quad \text{DO} \\
\{\{ P \land b \}\} \quad c_1 \quad \{\{ P \}\} \\
\text{END} \\
\{\{ P \land \neg b \}\}
\end{align*} \]

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

\[ \{\{ P \}\} \quad -\rightarrow \quad \{\{ P' \}\} \]
Small-Step Imp

Arithmetic expressions:

Reserved Notation " t '/ st '--a' t' "
(at level 40, st at level 39).

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').
Reserved Notation " t '/' st '-->' t '/' st "
(at level 40, st at level 39, t' at level 39).

Open Scope imp_scope.

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' -->
  TEST b THEN c1 ELSE c2 FI / st -->
  (TEST b' THEN c1 ELSE c2 FI) / st
| CS_IfTrue : forall st c1 c2,
  TEST BTrue THEN c1 ELSE c2 FI / st --> c1 / st
| CS_IfFalse : forall st c1 c2,
  TEST BFalse THEN c1 ELSE c2 FI / st --> c2 / st
| CS_While : forall st b c1,
  WHILE b DO c1 END / st -->
  (TEST b THEN c1;; WHILE b DO c1 END ELSE SKIP FI) / st

where " t '/' st '-->' t '/' st " := (cstep (t,st) (t',st')).