CIS 500: Software Foundations — Midterm II

November 8, 2016

Directions: This exam booklet contains both the standard and advanced track questions. Questions with no annotation are for *both* tracks. Other questions are marked "Standard Only" or "Advanced Only." *Do not waste time answering questions intended for the other track.*

Mark the box of the track you are following. (If you are following one track but want to move to the other now, please mark the box for the track you *want* to be on and write a note on this page telling us that we should switch you to this track.)

☐ Standard ☐ Advanced

NOTE: Throughout this exam, we use informal notation for Imp boolean expressions. For example, we write `X <> Y` instead of `BNot (BEq X Y)`.

1. (5 points)

   1. If `c` is equivalent to `c;;c`, then `c;;c` is equivalent to `c;;c;;c`.

      Answer: True

   2. If `c1;;c2` is equivalent to `c2;;c1`, then `c1` is equivalent to `c2`.

      Answer: False

   3. The following Hoare triple is valid.

      $\{\!\{\,\mathtt{True}\,\}\!\}$ `Y ::= X + 1; WHILE X <> 0 DO X ::= X - 1 END` $\{\!\{\,\mathtt{Y = X + 1}\,\}\!\}$

      Answer: False

4. The following two programs are equivalent:

```
WHILE X <> 0 DO SKIP END
```

and

```
WHILE X <> 0 DO Y ::= Y - 1 END
```

Answer: True

5. `normal_form` is a syntactic concept that is defined by looking at the form of a term, while `value` is a semantic concept that is defined by looking at how the term steps.

Answer: False

2. (10 points) Recall that the assertion P appearing in the `hoare_while` rule is called the *loop invariant*. For each loop shown below, circle *each* assertion that is a valid loop invariant. (There may be zero or more than one of them.)

1. `WHILE X<100 DO X ::= X+1 END`

   (a) $\boxed{X > 10}$
   (b) $X < 100$
   (c) $\boxed{X \leq 100}$
   (d) $\boxed{X > 100}$

2. `WHILE X>10 DO X ::= X+1 END`

   (a) $\boxed{X > 10}$
   (b) $X < 100$
   (c) $X \leq 100$
   (d) $\boxed{X \geq 5}$
   (e) $\boxed{\texttt{False}}$
   (f) $\boxed{\texttt{True}}$

3. `WHILE Y>0 DO Y ::= Y-1;; Z ::= Z+X END`

   (a) $\boxed{X > 10}$
   (b) $Y > 10$
   (c) $\boxed{W = Z + (X * Y)}$
   (d) $Z < Y$

3. **[Standard Only]** (12 points) Are the following Hoare logic rules of inference valid? Write *Valid* or *Invalid*. Additionally, if the rule is invalid, give a counterexample—i.e., a concrete P, Q, `c`, `st`, and `st'` such that (1) $\{\!\{ P \}\!\}$ `c` $\{\!\{ Q \}\!\}$ according to the proposed rule, (2) `c / st \\ st'`, (3) P holds of `st`, and (4) Q does not hold of `st'`.

   1. $$\frac{\{\!\{ P \}\!\}\ \texttt{c;;c}\ \{\!\{ P \}\!\}}{\{\!\{ P \}\!\}\ \texttt{c}\ \{\!\{ P \}\!\}} \quad (\texttt{hoare\_two\_to\_one})$$

      Invalid. For example, let P be `X = 1` and let `c` be `X ::= 1-X`.

   2. $$\frac{\{\!\{ P \wedge b \}\!\}\ \texttt{c}\ \{\!\{ Q \}\!\}}{\{\!\{ P \}\!\}\ \texttt{WHILE b DO c END}\ \{\!\{ Q \wedge \sim b \}\!\}} \quad (\texttt{hoare\_whilealt})$$

      Invalid. Consider this instance: $\{\!\{ X = 0 \}\!\}$ `WHILE X > 1 DO X := 1 END` $\{\!\{ X = 1 \wedge \sim(X > 1) \}\!\}$

   3. $$\frac{}{\{\!\{ P \}\!\}\ \texttt{X ::= a}\ \{\!\{ P[X \mapsto a] \}\!\}} \quad (\texttt{hoare\_assn\_forward})$$

      Invalid.

      Consider this instance. $\{\!\{ X = 0 \}\!\}$ `X := X + 1` $\{\!\{ X + 1 = 0 \}\!\}$

4. **[Advanced Only]** (13 points) The Imp command `WHILE True DO SKIP END` never terminates. Write a careful, informal proof of this fact. In other words, prove:

$$\forall \texttt{st st}',\ \sim(\texttt{WHILE True DO SKIP END/st} \Downarrow \texttt{st}')$$

Answer: Let `st` and `st'` be arbitrary. Suppose that there is some evaluation
`WHILE True DO SKIP END/st` $\Downarrow$ `st'`.
We will prove by induction that this evaluation is impossible.
By the form of the command, there are just two cases to consider

1. `(WHILE True DO SKIP END) / st` $\Downarrow$ `st'` by rule `E_WhileEnd`, with `st' = st` and `beval st True = false`. However, the evaluation of `True` is `true`, which cannot equal `false`, so this case is impossible.

2. `(WHILE True DO SKIP END) / st` $\Downarrow$ `st'` by rule `E_WhileLoop`, with `beval st True = true` and `SKIP / st` $\Downarrow$ `st1` and `(WHILE True DO SKIP END) / st1` $\Downarrow$ `st'`. By inversion of the evaluation `SKIP / st` $\Downarrow$ `st1`, we know that `st1` $=$ `st`. We know by induction that the subevaluation `(WHILE True DO SKIP END) / st` $\Downarrow$ `st'` is impossible, so this case also cannot occur.

5. (15 points) Recall the definition of what it means for a term `t` to be a *normal form* of a relation `R`.

```
Definition normal_form {X:Type} (R:relation X) (t:X) : Prop :=
  ~ 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 `astep` relation (i.e., for which the proposition `normal_form (astep st) a` holds for all states `st`).

   Answer: For any natural number `n`*,* `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 `cstep` relation.

   Answer: The only Imp program in normal form is `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 in normal form or we can take a step.*

5. Does strong progress imply termination?

   Answer: No

6. (10 points) Suppose we extend the Imp language with a simple mechanism for thowing and catching exceptions. The new command `THROW` throws an exception, while the command `TRY c1 CATCH c2 END` runs `c1` and—if `c1` terminates normally—simply terminates in the same final state; if `c1` throws an exception, on the other hand, `TRY c1 CATCH c2 END` will then run `c2` (starting from the state in which the exception was thrown in `c1`). For example:

- `TRY X ::= 5 CATCH X ::= 42 END` terminates normally with `X = 5`
- `(TRY X ::= 5 CATCH X ::= 42 END);; THROW` raises an exception with `X = 5`
- `TRY X ::= 5;; THROW CATCH X ::= X+1 END` terminates normally with `X = 6`
- `TRY X ::= 5;; THROW CATCH X ::= X+1;; THROW END` raises an exception with `X = 6`
- `TRY (TRY X ::= 5;; THROW CATCH X ::= X+1;; THROW) CATCH X ::= X+1 END` terminates normally with `X = 7`

The syntax of Imp is extended as follows:

```
Inductive com : Type :=
  | CSkip : com
  | CStop : com                 (* <-- new *)
  | CTry : com -> com -> com    (* <-- new *)
  | CAss : id -> aexp -> com
  | CSeq : com -> com -> com
  | CIf : bexp -> com -> com -> com
  | CWhile : bexp -> com -> com.

Notation "'THROW'" :=
  CStop.
Notation "'TRY' c1 'CATCH' c2 'END'" :=
  (CTry c1 c2) (at level 80, right associativity).
```

To define the evaluation relation, we need to distinguish between (1) normal termination of a command and (2) raising an exception. We introduce the following datatype for this purpose:

```
Inductive status : Type :=
  | SNormal : status
  | SThrow : status.
```

The evaluation relation is extended to include a `status`: we write `c / st \\ s / st'` to mean that running command `c` in state `st` terminates with status `s` and final state `st'`.

Intuitively, `c / st \\ s / st'` means that, if `c` is started in state `st`, then it terminates in state `st'` and the status `s` signals either that an exception has been raised (`s = SThrow`) or that execution terminated normally (`s = SNormal`).

- If the command is `SKIP`, then the state doesn't change and execution of any enclosing loop terminates normally.
- If the command is `THROW`, the state stays unchanged but we signal an exception.
- If the command is an assignment, then we update the binding for that variable in the state accordingly and signal that execution terminates normally.
- If the command is of the form `IFB b THEN c1 ELSE c2 FI`, then execution proceeds as in the original semantics of Imp, except that we also propagate the status from the execution of whichever branch was taken.
- If the command is a sequence `c1 ;; c2`, we first execute `c1`. If this terminates with status `SThrow`, we skip the execution of `c2` and terminate the whole command with status `SThrow`; the resulting state is the one resulting from `c1`. Otherwise, we execute `c2` as usual and yield its final state and status.
- If the command is `TRY c1 CATCH c2 END`, we begin by executing `c1`; if it terminates normally, then the whole `TRY` terminates in the same state and with normal status; if it terminates with an exception, we execute `c2` (in the state resulting from `c1`) and yield its final state and status.
- Finally, for a loop of the form `WHILE b DO c END`, when `b` evaluates to true, we execute `c` and check the status it returns. If it is `SNormal`, execution proceeds as in the original semantics. Otherwise, we stop the execution of the loop and yield status `SThrow`.

Based on the above description, complete the formal definition of the `ceval` relation on the following page.

```
Inductive ceval : com -> state -> status -> state -> Prop :=
  | E_Skip : forall st,
      CSkip / st \\ SNormal / st
  | E_Stop : forall st,
      CStop / st \\ SThrow / st
  | E_Ass  : forall st a1 n x,
      aeval st a1 = n ->
      (x ::= a1) / st \\ SNormal / (t_update st X n)
  | E_If : forall c1 c2 b st st' s,
      (if beval st b then c1 else c2) / st \\ s / st' ->
      (IFB b THEN c1 ELSE c2 FI) / st \\ s / st'
  | E_SeqContinue : forall c1 c2 st st' st'' s,
      c1 / st \\ SNormal / st' ->
      c2 / st' \\ s / st'' ->
      (c1 ;; c2) / st \\ s / st''
  | E_SeqStop : forall c1 c2 st st',
      c1 / st \\ SThrow / st' ->
      (c1 ;; c2) / st \\ SThrow / st'

  | E_TryContinue : forall c1 c2 st st',
      c1 / st \\ SNormal / st' ->
      (TRY c1 CATCH c2 END) / st \\ SNormal / st'
  | E_TryStop : forall c1 c2 st st' st'' s,
      c1 / st \\ SThrow / st' ->
      c2 / st' \\ s / st'' ->
      (TRY c1 CATCH c2 END) / st \\ s / st''
  | E_WhileEnd : forall c b st,
      beval st b = false ->
      (WHILE b DO c END) / st \\ SNormal / st
  | E_WhileContinue : forall c b st st' st'' s,
      beval st b = true ->
      c / st \\ SNormal / st' ->
      (WHILE b DO c END) / st' \\ s / st'' ->
      (WHILE b DO c END) / st \\ s / st''
  | E_WhileStop : forall c b st st',
      beval st b = true ->
      c / st \\ SThrow / st' ->
      (WHILE b DO c END) / st \\ SThrow / st'
```

7. (9 points) *(Continuation of previous problem.)* Now we need to extend Hoare logic appropriately to deal with the possibility of exceptions. We begin by introducing *Hoare quads*, which are just like Hoare triples but have one additional assertion. The Hoare quad `{{P}} c {{Q}} {{R}}` is read "If command `c` is run in a state satisfying `P`, then (1) if it terminates normally, the final state will satisfy `Q`, and (2) if it raises an exception (that is not caught by an internal `TRY`), the final state will satisfy `R`." In Coq:

```
Definition hoare_quad
           (P:Assertion) (c:com) (Q:Assertion) (R:Assertion) : Prop :=
  forall st st' s,
     c / st \\ s / st'  ->
     P st  ->
     (s = SNormal /\ Q st') \/ (s = SThrow /\ R st').

Notation "{{ P }} c {{ Q }} {{ R }}" :=
  (hoare_quad P c Q R) (at level 90, c at next level)
  : hoare_spec_scope.
```

For each of the following Hoare quads state whether it is valid or invalid. If it is invalid, give a counterexample.

1. $\{\!\{$ `True` $\}\!\}$ `TRY (X::=1;; THROW) CATCH THROW END` $\{\!\{$ `False` $\}\!\}$ $\{\!\{$ `X = 1` $\}\!\}$ *Answer: valid*
2. $\{\!\{$ `True` $\}\!\}$ `IFB Y=1 THEN THROW ELSE X::=2 FI` $\{\!\{$ `X = 2` $\}\!\}$ $\{\!\{$ `Y = 1` $\}\!\}$ *Answer: valid*
3. $\{\!\{$ `Z = 10` $\}\!\}$ `WHILE Z>0 DO (IFB Z=1 THEN THROW ELSE Z::=Z-1 FI) END` $\{\!\{$ `Z = 1` $\}\!\}$ $\{\!\{$ `Z = 0` $\}\!\}$
   Answer: invalid

8. **[Advanced Only]** (9 points) *(Continuation of previous problem.)* Here are the Hoare rules for `SKIP`, `THROW`, assignment, and conditionals.

$$\{\!\{ P \}\!\}\ \texttt{SKIP}\ \{\!\{ P \}\!\}\ \{\!\{ R \}\!\} \quad (\texttt{hoare\_skip})$$

$$\{\!\{ P \}\!\}\ \texttt{THROW}\ \{\!\{ Q \}\!\}\ \{\!\{ P \}\!\} \quad (\texttt{hoare\_stop})$$

$$\frac{}{\{\!\{ \texttt{assn\_sub X a } Q \}\!\}\ \texttt{X := a}\ \{\!\{ Q \}\!\}\ \{\!\{ R \}\!\}} \quad (\texttt{hoare\_asgn})$$

$$\frac{\begin{array}{c}\{\!\{ P \wedge b \}\!\}\ \texttt{c1}\ \{\!\{ Q \}\!\}\ \{\!\{ R \}\!\} \\ \{\!\{ P \wedge \sim b \}\!\}\ \texttt{c2}\ \{\!\{ Q \}\!\}\ \{\!\{ R \}\!\}\end{array}}{\{\!\{ P \}\!\}\ \texttt{IFB b THEN c1 ELSE c2 FI}\ \{\!\{ Q \}\!\}\ \{\!\{ R \}\!\}} \quad (\texttt{hoare\_if})$$

Write down appropriate rules for sequencing, `TRY`, and `WHILE`. (Just give the rules—no need to prove them!)

Answer: (Note that the rules are given here in Coq syntax here, while we asked you for informal inference rule syntax.)

```
Theorem hoare_seq : forall P Q R S c1 c2,
     {{Q}} c2 {{R}} {{S}} ->
     {{P}} c1 {{Q}} {{S}} ->
     {{P}} c1;;c2 {{R}} {{S}}.

Lemma hoare_try : forall P Q S1 S2 c1 c2,
  {{P}} c1 {{Q}} {{S1}} ->
```

```
      {{S1}} c2 {{Q}} {{S2}} ->
      {{P}} TRY c1 CATCH c2 END {{Q}} {{S2}}.

    Lemma hoare_while : forall P S b c,
      {{fun st => P st /\ bassn b st}} c {{P}} {{S}} ->
      {{P}} WHILE b DO c END {{fun st => P st /\ ~ (bassn b st)}} {{S}}.
```

9. (9 points) The Fibonacci function can be defined as follows:

$$\texttt{fib}(n) = \begin{cases} 1 & \text{if } n = 0 \\ 1 & \text{if } n = 1 \\ \texttt{fib}(n-1) + \texttt{fib}(n-2) & \text{otherwise} \end{cases}$$

(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 first few Fibonacci numbers are: 1, 1, 2, 3, 5, 8, 13, ...

The following Imp program computes the `n`th Fibonacci number and leaves it in the variable `Y`.

```
X ::= 1;;
Y ::= 1;;
Z ::= 1;;
WHILE X <> n+1 DO
  T ::= Z;;
  Z ::= Z + Y;;
  Y ::= T;;
  X ::= X + 1
END
```

1. Explain, in English, the roles of the variables `X`, `Y`, `Z`, and `T` in this program.

   Answer: `X` *counts from 1 up to* `n+1`*,* `Y` *holds the* `(X-1)`*th Fibonacci number at the end of each loop,* `Z` *holds the* `X`*th Fibonacci number at the end of each loop,* `T` *is used for temporary storage*

2. Suppose we want to prove the Hoare triple ⦃`True`⦄ `fibc` ⦃`Y` = `fib n`⦄, where `fibc` is the Imp program above.

   Write down a suitable loop invariant for this proof.

   Answer: `Z = fib X and Y = fib (pred X)`

10. **[Standard Only]** (10 points) On the next page, 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 *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 `->>`.

The implication steps in your decoration may rely (silently) on all the usual rules of natural-number arithmetic. You may also assume the following equations about `fib`:

```
  fib 0 = 1
  fib 1 = 1
  fib (n+2) = fib (n+1) + fib n
```

The rules of Hoare logic and the rules for well-formed decorated programs can be found on pages 3 and 4 of the handout.

```
{{ True }} ->>
{{ 1 = fib 1 /\ 1 = fib (pred 1) }}
X ::= 1;;
{{ 1 = fib X /\ 1 = fib (pred X) }}
Y ::= 1;;
{{ 1 = fib X /\ Y = fib (pred X) }}
Z ::= 1;;
{{ Z = fib X /\ Y = fib (pred X) }}
WHILE X <> n+1 DO
  {{ Z = fib X /\ Y = fib (pred X) /\ X <> n+1 }} ->>
  {{ Z+Y = fib (X+1) /\ Z = fib (pred (X+1)) }}
  T ::= Z;;
  {{ Z+Y = fib (X+1) /\ T = fib (pred (X+1)) }}
  Z ::= Z + Y;;
  {{ Z = fib (X+1) /\ T = fib (pred (X+1)) }}
  Y ::= T;;
  {{ Z = fib (X+1) /\ Y = fib (pred (X+1)) }}
  X ::= X + 1;;
  {{ Z = fib X /\ Y = fib (pred X) }}
END
{{ Z = fib X /\ Y = fib (pred X) /\ ~(X <> n+1) }} ->>
{{ Y = fib n }}
```

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.

Notation "'SKIP'" :=
  CSkip.
Notation "l '::=' a" :=
  (CAss l a) (at level 60).
Notation "c1 ;; c2" :=
  (CSeq c1 c2) (at level 80, right associativity).
Notation "'WHILE' b 'DO' c 'END'" :=
  (CWhile b c) (at level 80, right associativity).
Notation "'IFB' e1 'THEN' e2 'ELSE' e3 'FI'" :=
  (CIf e1 e2 e3) (at level 80, right associativity).
```

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 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 :=
  forall 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 :=
  forall st, P st -> Q st.

Notation "P ->> Q" := (assert_implies P Q) (at level 80).
```

(ASCII `->>` is typeset as a hollow arrow in the rules below.)

Hoare logic rules

$$\frac{}{\{\!\{\, \texttt{assn\_sub X a } Q \,\}\!\}\ \texttt{X := a}\ \{\!\{\, Q \,\}\!\}}\ (\texttt{hoare\_asgn})$$

$$\frac{}{\{\!\{\, P \,\}\!\}\ \texttt{SKIP}\ \{\!\{\, P \,\}\!\}}\ (\texttt{hoare\_skip})$$

$$\frac{\begin{array}{c}\{\!\{\, P \,\}\!\}\ \texttt{c1}\ \{\!\{\, Q \,\}\!\} \\ \{\!\{\, Q \,\}\!\}\ \texttt{c2}\ \{\!\{\, R \,\}\!\}\end{array}}{\{\!\{\, P \,\}\!\}\ \texttt{c1;; c2}\ \{\!\{\, R \,\}\!\}}\ (\texttt{hoare\_seq})$$

$$\frac{\begin{array}{c}\{\!\{\, P \wedge b \,\}\!\}\ \texttt{c1}\ \{\!\{\, Q \,\}\!\} \\ \{\!\{\, P \wedge \sim b \,\}\!\}\ \texttt{c2}\ \{\!\{\, Q \,\}\!\}\end{array}}{\{\!\{\, P \,\}\!\}\ \texttt{IFB b THEN c1 ELSE c2 FI}\ \{\!\{\, Q \,\}\!\}}\ (\texttt{hoare\_if})$$

$$\frac{\{\!\{\, P \wedge b \,\}\!\}\ \texttt{c}\ \{\!\{\, P \,\}\!\}}{\{\!\{\, P \,\}\!\}\ \texttt{WHILE b DO c END}\ \{\!\{\, P \wedge \sim b \,\}\!\}}\ (\texttt{hoare\_while})$$

$$\frac{\begin{array}{c}\{\!\{\, P' \,\}\!\}\ \texttt{c}\ \{\!\{\, Q' \,\}\!\} \\ P \twoheadrightarrow P' \\ Q' \twoheadrightarrow Q\end{array}}{\{\!\{\, P \,\}\!\}\ \texttt{c}\ \{\!\{\, Q \,\}\!\}}\ (\texttt{hoare\_consequence})$$

$$\frac{\begin{array}{c}\{\!\{\, P' \,\}\!\}\ \texttt{c}\ \{\!\{\, Q \,\}\!\} \\ P \twoheadrightarrow P'\end{array}}{\{\!\{\, P \,\}\!\}\ \texttt{c}\ \{\!\{\, Q \,\}\!\}}\ (\texttt{hoare\_consequence\_pre})$$

$$\frac{\begin{array}{c}\{\!\{\, P \,\}\!\}\ \texttt{c}\ \{\!\{\, Q' \,\}\!\} \\ Q' \twoheadrightarrow Q\end{array}}{\{\!\{\, P \,\}\!\}\ \texttt{c}\ \{\!\{\, Q \,\}\!\}}\ (\texttt{hoare\_consequence\_post})$$

Decorated programs

1. `SKIP` is locally consistent if its precondition and postcondition are the same:

   ```
   {{ P }}
   SKIP
   {{ P }}
   ```

2. The sequential composition of `c1` and `c2` is locally consistent (with respect to assertions `P` and `R`) if `c1` is locally consistent (with respect to `P` and `Q`) and `c2` is locally consistent (with respect to `Q` and `R`):

   ```
   {{ P }}
   c1;;
   {{ Q }}
   c2
   {{ R }}
   ```

3. An assignment is locally consistent if its precondition is the appropriate substitution of its postcondition:

   ```
   {{ P [X |-> a] }}
   X ::= a
   {{ P }}
   ```

4. 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 /\ b` and `P /\ ~b` and if its "then" branch is locally consistent (with respect to `P /\ b` and `Q`) and its "else" branch is locally consistent (with respect to `P /\ ~b` and `Q`):

   ```
   {{ P }}
   IFB b THEN
     {{ P /\ b }}
     c1
     {{ Q }}
   ELSE
     {{ P /\ ~b }}
     c2
     {{ Q }}
   FI
   {{ Q }}
   ```

5. A while loop with precondition `P` is locally consistent if its postcondition is `P /\ ~b` and if the pre- and postconditions of its body are exactly `P /\ b` and `P`:

```
{{ P }}
WHILE b DO
  {{ P /\ b }}
  c1
  {{ P }}
END
{{ P /\ ~b }}
```

6. A pair of assertions separated by `->>` is locally consistent if the first implies the second (in all states):

```
{{ P }} ->>
{{ 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') (at level 40).
```

Small Step Semantics

```
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_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').

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

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')).
```