CIS 500 — Software Foundations

Midterm II

(Standard and advanced versions together)

November 11, 2014

Answer key

(8 points)

1. **Hoare triples**

Which of the the Hoare triples below are valid? If a triple is valid, circle the rules of Hoare logic that are *necessary* to justify the validity of that triple. You may need to circle more than one rule for a given triple, but do not circle a particular rule if the triple can be justified without it. Otherwise, if the triple is invalid, circle the last bullet.

For reference, the rules of Hoare logic are given in the Appendix, starting on page 14.

(a) $\{\{ 0 \le 3 + 4 \}\}$ `X ::= 3 + 4` $\{\{ 0 \le X \}\}$

- `hoare_asgn`
- `hoare_skip`
- `hoare_while`
- `hoare_consequence`
- *Not a valid Hoare Triple*

`hoare_asgn`

(b) $\{\{ X = X + 1 \}\}$ `X ::= X + 1` $\{\{ True \}\}$

- `hoare_asgn`
- `hoare_skip`
- `hoare_while`
- `hoare_consequence`
- *Not a valid Hoare Triple*

`hoare_consequence` and `hoare_asgn`

(c) $\{\{ True \}\}$ `X ::= X + 1` $\{\{ X = X + 1 \}\}$

- `hoare_asgn`
- `hoare_skip`
- `hoare_while`
- `hoare_consequence`
- *Not a valid Hoare Triple*

Not a valid Hoare Triple

(d) $\{\{ True \}\}$ `WHILE BTrue DO SKIP END` $\{\{ False \}\}$

- `hoare_asgn`
- `hoare_skip`

- `hoare_while`
- `hoare_consequence`
- *Not a valid Hoare Triple*

`hoare_while`, `hoare_skip` and `hoare_consequence`

Grading scheme: 2 points for each bullet. 1 point for getting valid/invalid correct and an additional point for marking the correct rules. No points were awarded for answers that were left blank: this problem asked how to prove these triples with the Hoare rules.

(15 points)

2. **Properties of Imp relations**

Which of the following propositions about Imp are provable in Coq? (You may reason using the axiom `functional_extensionality`, if needed.) Circle True or False. If the property is not provable, explain why or provide a counterexample.

For reference, the relations `ceval` ($\texttt{c} \,/\texttt{st} \Downarrow \texttt{st}'$), `cequiv`, and Hoare triples ($\{\!\{ P \}\!\}\ \texttt{c}\ \{\!\{ Q \}\!\}$) appear on pages 14 and 15.

(a) $\exists \texttt{c},\ \forall \texttt{st st}',\ \sim(\texttt{c}/\texttt{st} \Downarrow \texttt{st}')$

Answer: true

(b) $\forall \texttt{c st st}',\ (\texttt{c}/\texttt{st} \Downarrow \texttt{st}')$

Answer: false If `c = WHILE TRUE DO SKIP END`, then there are no finite derivations of its evaluation (for any states).

(c) $\forall \texttt{c st st1 st2},\ (\texttt{c}/\texttt{st} \Downarrow \texttt{st1}) \rightarrow (\texttt{c}/\texttt{st} \Downarrow \texttt{st2}) \rightarrow \texttt{st1} = \texttt{st2}$

Answer: true

(d) $\forall \texttt{c st st1 st2},\ (\texttt{c}/\texttt{st1} \Downarrow \texttt{st}) \rightarrow (\texttt{c}/\texttt{st2} \Downarrow \texttt{st}) \rightarrow \texttt{st1} = \texttt{st2}$

Answer: false Consider when `c = X ::= 3, st1 = empty_state, st2 = extend empty_state X 4, st = extend empty_state X 3`

(e) $\forall P\ Q\ \texttt{c1 c2},\ \{\!\{ P \}\!\}\ \texttt{c1}\ \{\!\{ Q \}\!\} \rightarrow \{\!\{ P \}\!\}\ \texttt{c2}\ \{\!\{ Q \}\!\} \rightarrow \texttt{cequiv c1 c2}$

Answer: false If P is $True$ and Q is $False$, then `c1` and `c2` can be any commands.

(f) $\forall P\ Q\ \texttt{c1 c2},\ \texttt{cequiv c1 c2} \rightarrow (\{\!\{ P \}\!\}\ \texttt{c1}\ \{\!\{ Q \}\!\} \leftrightarrow \{\!\{ P \}\!\}\ \texttt{c2}\ \{\!\{ Q \}\!\})$

Answer: true

Grading scheme: 2 points for true, 1 point for false, 2 points for counterexample

(10 points)

3. **Decorated Programs**

Recall the factorial function, written in Coq:

```
 Fixpoint fact (n:nat) : nat :=
   match n with
   | O => 1
   | S n' => n * (fact n')
   end.
```

The following Imp program computes the factorial of X and places the answer into Y.

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

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. Use informal notations for mathematical formulae and assertions (and abbreviate `fact x` with `!x`, but 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 the following facts, as well as the usual rules of arithmetic:

- `minus_n_O : forall n, n - 0 = n`
- `mult_assoc : forall m n p, m * (n * p) = (m * n) * p`
- `mult_1_r  : forall m, m * 1 = m`

The Hoare rules and the rules for well-formed decorated programs are provided on pages 15 and 16, for reference.

```
  {{ X = m }} ->>
  {{ 1 * X! = m! }}
  Y ::= 1;;
  {{ Y * X! = m! }}
  WHILE X <> 0
  DO {{ Y * X! = m! /\ X <> 0 }} ->>
     {{ Y * X * (X - 1)! = m! }}
     Y ::= Y * X;;
     {{ Y * (X - 1)! = m! }}
      X ::= X - 1
     {{ Y * X! = m! }}
END
  {{ Y * X! = m! /\ ~(X <> 0) }} ->>
  {{ Y = m! }}
```

Grading scheme:

- *1 point per implication*
- *3 points for correct "back propagation" of the mechanical parts of the annotation process*
- *4 points for the loop invariant*

We permited loop invariants of the form `X! = m!  / Y`*, though technically we haven't defined division in Coq.*

(20 points)

4. **Imp Extensions**

In this exercise, consider extending Imp with for loops, similar to those found in many other imperative programming language. Our concrete syntax for these loops might look something like this:

```
FOR ( initialization ;;  condition ;; increment )
   loopbody
END
```

The initialization command is run before the loop begins. The condition is some boolean expression, and terminates the loop if false. The increment command is performed exactly once every time at the end of each loop iteration.

To formalize the extended language, we first add a clause to the definition of commands with the four components of this new command.

```
Inductive com : Type :=
  ...
  | CFor : com -> bexp -> com -> com -> com.
```

(For simplicitly in the exam, we will not define a Coq notation for this command.) For example, we might represent the following for loop, written in the concrete syntax,

```
FOR (X ::= 0 ;;  X <= 10 ;; X ::= X + 1)
  Y ::= Y * X
END
```

as the following Coq expression:

```
CFor (X ::= ANum 0)                  (* initialization *)
     (BLe (AId X) (ANum 10))         (* condition      *)
     (X ::= APlus (AId X) (ANum 1))  (* increment      *)
     (Y ::= AMult (AId Y) (AId X))   (* loopbody       *)
```

(Problem continues on the next page.)

(a) Refer to the definition of `ceval` (page 14) for the evaluation relation of Imp. What rule(s) must be added to this definition to formalize the behavior of `CFor`? Write out the additional rule(s) in formal Coq notation.

```
Inductive ceval : com -> state -> state -> Prop :=
```

Answer:

```
| E_ForTrue : forall b st1 st2 st3 st4 st5 cinit cstep cbody,
  cinit / st1 || st2 ->
  beval st2 b = true ->
  cbody / st2 || st3 ->
  cstep / st3 || st4 ->
  (CFor SKIP b cstep cbody) / st4 || st5 ->
  (CFor cinit b cstep cbody) / st1 || st5
| E_ForFalse : forall b st1 st2 cinit cstep cbody,
  cinit / st1 || st2 ->
  beval st2 b = false ->
  (CFor cinit b cstep cbody) / st1 || st2
```

Grading scheme: 8 points total. This was a difficult problem. Common errors included:

- *evaluating* `cinit` *too many times.*
- *not evaluating* `cinit`
- *not evaluating* `cstep`
- *not evaluating* `cbody`
- *not looping*
- *not terminating the loop*
- *wrong order of evaluation, or incorrect sequencing of states*

(b) For each purported theorem about Imp with `CFor` commands below, write either "provable" if the claim is provable, or give a brief (one sentence) explanation, with a counterexample if possible, of why the claim is not provable. For your reference, the definition of `cequiv`, which remains unchanged from standard Imp, is found on page 14.

i.
```
Theorem thm1 : forall cincr,
  cequiv SKIP (CFor SKIP BTrue cincr SKIP).
```

Answer: Not provable. The for loop is an infinite loop.

ii.
```
Theorem thm2 :
  cequiv   (CFor SKIP BTrue SKIP SKIP)
           (WHILE BTrue DO SKIP).
```

Answer: Provable. Both commands are infinite loops.

iii.
```
Theorem thm3 : forall cinit bcond cincr cstep,
  cequiv  (CFor cinit bcond cincr cstep)
          (cinit ;; CFor SKIP bcond (cincr ;; cstep) SKIP).
```

Answer: Not Provable, it should be `(cstep ;; cincr)` in the second command.

Grading scheme: 3 pts for parts (a) and (b) each. Part (c) was not graded as it was trickier than intended.

(c) Write a Hoare proof rule for the `CFor` command. (For reference, the standard Hoare rules for Imp are provided on page 15.)

Your rule must be sound. It should also be as precise as possible.

$$\frac{\{\!\{ P \}\!\}\ \texttt{ci}\ \{\!\{ Q \}\!\} \quad \{\!\{ Q \wedge b \}\!\}\ \texttt{cb}\ \{\!\{ R \}\!\} \quad \{\!\{ R \}\!\}\ \texttt{cs}\ \{\!\{ Q \}\!\}}{\{\!\{ P \}\!\}\ \texttt{CFor ci b cs cb}\ \{\!\{ Q \wedge \sim b \}\!\}} \quad (\texttt{hoare\_for})$$

Grading scheme: 6 points. Common errors included:

- *Not using Q for the post condition of* `cs` *(2 pts)*
- *Missing* $\sim b$ *in the postcondition*
- *Q instead of R*
- *P instead of R*
- *Other errors at discretion*

(8 points)

5. **Program approximation**

In this question, we define an assymmetric variant of program equivalence we call *program approximation*. We say that program c1 *approximates* program c2 when, for each of the initial states for which c1 terminates, c2 also terminates and produces the same final state. Formally, program approximation can be defined as follows:

```
Definition capprox (c1 c2 : com) : Prop :=
  forall (st st' : state),
    (c1 / st || st') -> (c2 / st || st').
```

For example, the program c1 = WHILE X <> 1 DO X := X - 1 END approximates the program c2 = X := 1, but c2 does not approximate c1 because c1 does not terminate when X = 0. If two programs approximate eachother, then they are equivalent.

(a) Find two programs, c3 and c4, such that neither approximates the other. Your programs should be short (3 lines max).

```
c3 = X ::= 1
c4 = X ::= 2
```

Grading scheme: 4 points

(b) Find a program cmin that approximates every other program. Formally, the proposition forall c', capprox cmin c' should be provable. (Again, 3 lines max).

```
cmin = WHILE true DO SKIP END
```

Grading scheme: 4 points

(9 points)

6. **[Standard] Hoare Logic**

Are the following Hoare logic rules of inference valid? Write *Valid* or *Invalid*. Additionally, if the rule is invalid, give a counterexample.

(a)

$$\frac{\{\!\{\, P \,\}\!\}\ \texttt{c}\ \{\!\{\, Q \,\}\!\}}{\{\!\{\, P \,\}\!\}\ \texttt{IF b THEN c ELSE c FI}\ \{\!\{\, Q \,\}\!\}}\quad (\texttt{hoare\_bothif})$$

Valid

Grading scheme: 3 points

(b)

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

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

Grading scheme: 3pts. 1 for saying invalid, 2 for counterexample. A counter example is some triple $\{\!\{\, P \,\}\!\}$ `c` $\{\!\{\, Q \,\}\!\}$ *(including definitions of P,* `c`*, and Q) that this rule derives, but is not valid according to the definitions.*

(c)

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

Invalid.

Consider this instance. $\{\!\{\, X = 0 \,\}\!\}$ `X := X + 1` $\{\!\{\, X + 1 = 0 \,\}\!\}$ *Grading scheme: 3pts. 1 for saying invalid, 2 for counterexample*

(9 points)

7. **[Advanced] Informal proof**

Recall that the command `WHILE BTrue DO SKIP END` is an infinite loop.
Write a careful, informal proof of this fact. In other words, prove:

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

Answer: Let `st` and `st'` be arbitrary. Suppose that there is some evaluation `WHILE BTrue 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

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

(b) `(WHILE BTrue DO SKIP END) / st` $\Downarrow$ `st'` by rule `E_WhileLoop`, with `beval st BTrue = true` and `SKIP / st` $\Downarrow$ `st1` and `(WHILE BTrue 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 BTrue DO SKIP END) / st` $\Downarrow$ `st'` is impossible, so this case also cannot occur.

Grading scheme: Common errors included:

- *Failure to use induction (4pts)*
- *Incorrect use of the IH (2pts)*
- *Missing inversion on the evaluation of SKIP (1pt)*
- *Other errors at discretion*

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

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

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

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

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

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

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

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

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

Decorated programs

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

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

(b) 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 }}
```

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

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

(e) 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 }}
```

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

```
{{ P }} ->>
{{ P’ }}
```