CIS 500 — Software Foundations

Midterm I

(Advanced version)

March 27, 2013

Name: ____

Pennkey (e.g. `bcpierce`): ____

Scores:

1		8
2		12
3		12
4		15
5		18
6		15
Total:		80

1. (8 points) Indicate whether or not each of the following Hoare triples is valid by writing either "valid" or "invalid." Also, for those that are invalid, give a counter-example. The definition of Hoare triples is given on page 10, for reference.

(a)
```
{{ X=n /\ Y=m /\ Z=o }}
X ::= Y;
Y ::= Z;
Z ::= X
{{ X=n /\ Y=o /\ Z=m }}
```

(b)
```
{{ X=1 \/ Z=0 }}
IFB  Z=0  THEN
     X ::= 0
THEN
     X ::= 1 - X
FI;
Z ::= 1
{{ X=0 /\ Z=1 }}
```

(c)
```
{{ True }}
WHILE X > 0 DO
   X ::= X - 1;
   Y ::= X
END
{{ X=0 /\ Y=0 }}
```

(d)
```
{{ X>0 }}
WHILE X > 0 DO
   X ::= X + 1;
   Z ::= X - 1
END
{{ Z=X }}
```

2. (12 points) Given the following programs, group together those that are equivalent in Imp by drawing boxes around their names. For example, if you think programs a through h are all equivalent to each other, but not to i, your answer should look like this: $\boxed{a, b, c, d, e, f, g, h}$ $\boxed{i}$.

The definition of program equivalence is repeated on page 10, for reference.

(a)
```
X ::= Y;
Y ::= Z;
X ::= 0
```

(b)
```
IFB Y > 3 THEN
  X ::= 2 * Y
ELSE
  X ::= 2 * Y
FI;
Y ::= X
```

(c)
```
WHILE X > 0 DO
   X ::= 0;
   SKIP
END
```

(d)
```
WHILE X > 0 DO
  X ::= X * Y + 1
END
```

(e)
```
X ::= 0;
Y ::= Z
```

(f)
```
X ::= Y;
WHILE X > 0 DO
  Y ::= X + 1;
  X ::= X - 1
END;
X ::= Y
```

(g)
```
Y ::= Z;
WHILE X > 0 DO
  X ::= X - 1;
  Y ::= Z
END
```

(h)
```
WHILE X <> X DO
  X ::= X + 1
END;
X::=0
```

(i)
```
X ::= 2 * Y;
Y ::= 2 * Y
```

3. (12 points) In this question we consider extending Imp with `REPEAT` statements of the form

```
REPEAT c UNTIL b END
```

where `b` is a boolean expression, and `c` is a command. `REPEAT` behaves like `WHILE` except that the loop guard is checked *after* each execution of the body, with the loop repeating as long as the guard stays *false*. Because of this, the body will always execute at least once.

To formalize the extended language, we first add a clause to the definition of commands:

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

Notation "'REPEAT' e1 'UNTIL' b2 'END'" :=  (CRepeat e1 b2).
```

(a) Refer to the definition of `ceval` (page 10) for the evaluation relation of Imp. What rule(s) must be added to this definition to formalize the behavior of `REPEAT`? Write out the additional rule(s) in formal Coq notation.

(b) Write a Hoare proof rule for `REPEAT`.

Try to come up with a rule that is both sound and as precise as possible. For full credit, make sure your rule can be used to prove the following valid Hoare triple:

```
{{ Y <= m }}
REPEAT
  X ::= X + 1;
  IFB X <= m THEN Y ::= X ELSE SKIP END
UNTIL X > m END
{{ X > m /\ Y <= m }}
```

4. (15 points) Suppose we've defined a Coq function `sort` that sorts lists of numbers. The following Imp program performs an analogous (though simpler) task: it sorts the numbers stored in the variables `X`, `Y`, and `Z`.

```
  {{ X=m /\ Y=n /\ Z=o }}
WHILE X > Y \/ Y > Z DO
  IF X > Y THEN
    W := X;
    X := Y;
    Y := W
  ELSE
    SKIP
  FI;
  IF Y > Z THEN
    W := Y;
    Y := Z;
    Z := W
  ELSE
    SKIP
  FI
END
  {{ sort[m,n,o] = [X,Y,Z] }}
```

On the next page, add appropriate annotations 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, 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 for decorated programs (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 Hoare rules and the rules for well-formed decorated programs are provided on pages 11 and 12, for reference.

The implication steps in your decoration may rely (silently) on the following facts about `sort`:

- If `l1` is a permutation of `l2` (i.e., they have the same elements, but perhaps not in the same order), then `sort l1 = sort l2`.
- If each element of `l` is less than or equal to the following element, then `sort l = l`.

```
  {{ X=m /\ Y=n /\ Z=o }} ->>
  {{                                                                }}
WHILE X > Y \/ Y > Z DO
    {{                                                              }} ->>
    {{                                                              }}
  IF X > Y THEN
      {{                                                            }} ->>
      {{                                                            }}
    W := X;
      {{                                                            }}
    X := Y;
      {{                                                            }}
    Y := W
      {{                                                            }}
  ELSE
      {{                                                            }}
    SKIP
      {{                                                            }}
  FI;
    {{                                                              }}
  IF Y > Z THEN
      {{                                                            }} ->>
      {{                                                            }}
    W := Y;
      {{                                                            }}
    Y := Z;
      {{                                                            }}
    Z := W
      {{                                                            }}
  ELSE
      {{                                                            }}
    SKIP
      {{                                                            }}
  FI
    {{                                                              }}
END
  {{                                                                }} ->>
  {{ sort [m,n,o] = [X,Y,Z] }}
```

5. (18 points) The following program implements “slow multiplication” in Imp.

```
  {{ True }}
Y ::= 0;
Z ::= 0;
WHILE Y < n DO
  X ::= 0;
  WHILE X < m DO
    Z ::= Z + 1;
    X := X + 1
  END;
  Y ::= Y + 1
END
  {{ Z = n*m }}
```

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.

```
  {{ True }} ->>
  {{                                                                  }}
Y ::= 0;
  {{                                                                  }}
Z ::= 0;
  {{                                                                  }}
WHILE Y < n DO
    {{                                                                }} ->>
    {{                                                                }}
  X ::= 0;
    {{                                                                }}
  WHILE X < m DO
      {{                                                              }} ->>
      {{                                                              }}
    Z ::= Z + 1;
      {{                                                              }}
    X := X + 1
      {{                                                              }}
  END;
    {{                                                                }} ->>
    {{                                                                }}
  Y ::= Y + 1
    {{                                                                }}
END
  {{                                                                  }} ->>
  {{ Z = n*m }}
```

6. (15 points) Recall the Hoare logic rule for `WHILE` loops:

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

Write a careful informal proof of its correctness.

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

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

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

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

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

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

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

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

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

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' }}
```