CIS 500: Software Foundations — Midterm II

November 17, 2020

Name (printed): ______________________________

Username (PennKey login id): ______________________________

My signature below certifies that I have complied with the University of Pennsylvania's Code of Academic Integrity in completing this examination.

Signature: ______________________________ Date: ____________________

Directions:

- This exam contains both standard and advanced-track questions. Questions with no annotation are for *both* tracks. Other questions are marked "Standard Track Only" or "Advanced Track Only."

  Do not waste time (or confuse the graders) by answering questions intended for the other track. To make sure, please look for the questions for the other track as soon as you begin the exam and cross them out!

- Before beginning the exam, please write your PennKey (login ID) at the top of each even-numbered page (so that we can find things if a staple fails!).

Mark the box of the track you are following.

☐ Standard ☐ Advanced

$\boxed{1}$ (15 points)

Suppose we are given a command `c` and a desired postcondition `Q`. In general, there may be many preconditions `P` that make the Hoare triple `{{P}} c {{Q}}` valid. But it is a property of Hoare logic that, among all these, there will be one such `P` that is *weaker* than all the others—i.e., such that `P' ->> P` whenever `{{P'}} c {{Q}}` is valid.

For example, these are all valid triples,

```
{{ False }} X := X + 1 {{ X=2 }}
{{ X=1 /\ Y=2 }} X := X + 1 {{ X=2 }}
{{ X=1 }} X := X + 1 {{ X=2 }}
```

but `X=1` is the weakest precondition for this command and postcondition.

Complete the following triples with their weakest preconditions.

$\boxed{1.1}$
```
{{ P }} X := 4 {{ X * Y = 16 }}

P =
```

$\boxed{1.2}$
```
{{ P }} Z := X; X := Y; Y := Z {{ X = n /\ Y = m }}

P =
```

$\boxed{1.3}$
```
{{ P }} while Y >= 0 do Y := Y - X end {{ False }}

P =
```

$\boxed{1.4}$
```
{{ P }} while Y > 0 do X := X + 1; Y := Y - 1 end {{ X >= Y  }}

P =
```

$\boxed{1.5}$
```
{{ P }}
  C := 0;
  while C > 0 do
    X := X - 1;
    C := C + 2
{{ C = n * 2 }}

P =
```

$\boxed{2}$ (6 points)

Consider the following Hoare triple containing a while loop. Give a valid loop invariant P for this while loop that is strong enough to prove the post condition via application of the consequence rule. We've added some "partial decorations" to help you check the conditions that P must satisfy.

```
{{X = m /\ Y = n}}
  DONE := 0;
{{ P }}
  while DONE = 0 do
    {{ P /\ DONE = 0 }}
      if X > Y then
        T ::= Y;
        Y ::= X;
        X ::= T
      else
        DONE := 1
      end
    {{ P }}
  end
{{ P /\ ~(DONE = 0)}} ->>
{{X = min(m,n) /\ Y = max(m,n)}}
```

```
P  =
```

$\boxed{3}$ (15 points)

In this problem, we will be interested in Imp programs of a particular form: some initialization steps `c1`, followed by a while loop with body `c2`.

```
c1;
while b do
  c2
end
```

We'll "partially decorate" these programs with an initial precondition `I` and a loop invariant `P`:

```
{{ I }}
  c1;
{{ P }}
  while b do
    {{ P /\ b }}
      c2
    {{ P }}
  end
{{ P /\ ~b }}
```

Such a partially decorated program can fail to be valid for two reasons:

(a) the loop invariant can fail to be *established* by the initialization steps—that is, the triple `{{I}}c1{{P}}` can be invalid, and/or

(b) the loop invariant can fail to be *preserved* by the loop body—that is, the triple `{{P/\b}}c2{{P}}` can be invalid.

Below, we give several Imp programs and initial preconditions, and several candidate loop invariants for each. For each candidate, check *Establishment Fails* if establishment fails, *Preservation Fails* if preservation fails, and *Valid Invariant* if neither fails. (It is possible that both establishment and preservation fail; check both boxes in this case.)

$\boxed{3.1}$

```
{{ Y < X }}
  skip;
{{ P }}
  while Y <= X do
    {{ P /\ Y <= X }}
      if X = Y
        then Y := Y + 1
        else Y := Y + 3
      end
    {{ P }}
  end
{{ P /\ ~(Y <= X)

P =   Y < X
```

☐ Establishment Fails ☐ Preservation Fails ☐ Valid Invariant

$\boxed{3.2}$ `P =   Y < X + 1`

☐ Establishment Fails ☐ Preservation Fails ☐ Valid Invariant

$\boxed{3.3}$ `P =   Y < X + 2`

☐ Establishment Fails ☐ Preservation Fails ☐ Valid Invariant

$\boxed{3.4}$ `P =   Y < X + 3`

☐ Establishment Fails ☐ Preservation Fails ☐ Valid Invariant

$\boxed{3.5}$ `P =   Y < X + 4`

☐ Establishment Fails ☐ Preservation Fails ☐ Valid Invariant

3.6

```
{{ X = n /\ Y = n /\ Z = 0 }}
  skip;
{{ P }}
  while X > 0
    {{ P /\ X > 0 }}
      while Y > 0
        Z := Z + 1;
        Y := Y - 1
      end;
      X := X - 1
      Y := n
    {{ P }}
  end
{{ P /\ ~ (X > 0) }}
```

`P =   Y >= X`

☐ Establishment Fails ☐ Preservation Fails ☐ Valid Invariant

3.7 `P =   Y <= Z`

☐ Establishment Fails ☐ Preservation Fails ☐ Valid Invariant

3.8 `P =   Z = n * (n - X)`

☐ Establishment Fails ☐ Preservation Fails ☐ Valid Invariant

3.9 `P =   Y = n`

☐ Establishment Fails ☐ Preservation Fails ☐ Valid Invariant

3.10 `P =   Z = n * (n - Y)`

☐ Establishment Fails ☐ Preservation Fails ☐ Valid Invariant

```
{{ even X /\ even Y }}
  skip;
{{ P }}
  while X > 0 do
    {{ P /\ X > 0 }}
      Y := Y + 4;
      X := X - 2
    {{ P }}
  end
{{ P /\ ~ (X > 0) }}
```

$\boxed{3.11}$ `P =   even X`

☐ Establishment Fails ☐ Preservation Fails ☐ Valid Invariant

$\boxed{3.12}$ `P =   even Y`

☐ Establishment Fails ☐ Preservation Fails ☐ Valid Invariant

$\boxed{3.13}$ `P =   exists n, 4 * n = X`

☐ Establishment Fails ☐ Preservation Fails ☐ Valid Invariant

$\boxed{3.14}$ `P =   exists n, 4 * n = Y`

☐ Establishment Fails ☐ Preservation Fails ☐ Valid Invariant

$\boxed{3.15}$ `P =   Y <= X`

☐ Establishment Fails ☐ Preservation Fails ☐ Valid Invariant

$\boxed{4}$ (16 points)

Each of the following variations on the standard Hoare Logic rules is flawed in some way: it is either *unsound* (there are instances of the rule that are not valid Hoare triples) or *incomplete* (there are valid Hoare triples that are not provable if we substitute this rule in place of the standard one), or possibly both.

For instance, the following modified rule for `while` is incomplete.

```
{{P /\ b}} c {{P /\ b}} ->
{{P}} while b do c end {{P /\ ~ b}}.
```

because, if we replace the standard rule for `while` commands with this one, the following triple is not provable (just from the Hoare rules, without unfolding `hoare_triple`):

```
{{ 2 = 2 }}
  while X > 2 || X < 2 do
    x := 2
  end
{{ x = 2 }}
```

For each of the following rules, please say whether it is sound or unsound, and whether it is complete or incomplete. If they are unsound or incomplete, provide a counterexample in the same form as above. If the rule is both incomplete and unsound, give both counterexamples.

$\boxed{4.1}$

```
{{ True }} skip {{ True }}

[] Sound
[] Unsound...
      For example:

[] Complete
[] Incomplete...
      For example:
```

$\boxed{4.2}$

```
{{P}} c1 {{Q /\ b}} ->
{{P}} c2 {{Q /\ ~b}} ->
{{P}} if b then c1 else c2 end {{Q}}.

[] Sound
[] Unsound...
      For example:

[] Complete
[] Incomplete...
      For example:
```

$\boxed{4.3}$

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

[] Sound
[] Unsound...
      For example:
```

```
[] Complete
[] Incomplete...
     For example:
```

$\boxed{4.4}$

```
{{P}} c {{P}} ->
{{P}} while b do c end {{P /\ ~ b}}.

[] Sound
[] Unsound...
     For example:

[] Complete
[] Incomplete...
     For example:
```

$\boxed{5}$ (15 points)

Recall the definition of `Himp` — i.e., `Imp` extended with `havoc`, a nondeterministic variable assignment command. The command `havoc X` assigns an *arbitrary* number to the variable `X`, nondeterministically.

Since a given command from a given starting state can now reach many ending states, it is interesting to compare commands according to the sets of ending states they can produce. We say that command `c` *refines* command `c'` if, for every starting state, the set of possible ending states after running `c` is a subset of the set of ending states after running `c'`.

For example, the command

```
havoc X; X := X * 2
```

refines the command

```
havoc X
```

because the second can terminate with `X` set to any number whatsoever, while the first can only set `X` to even numbers.

A *counterexample* for the claim "command `c` refines command `c'`" consists of a particular starting state `st` and a particular ending state `st'` such that `st =[ c ]=> st'` holds, but `st =[ c' ]=> st'` does not.

Here are several pairs of HIMP commands. For each pair, answer whether the one on the left refines the one on the right, or the one on the right refines the one on the left, or both (i.e., the two commands produce the same sets of possible output states for every input state), or neither.

If you answer that one of them does not refine the other, please provide a counterexample.

$\boxed{5.1}$

```
havoc X
```

```
havoc X;
X := X + 1
```

(i) Does the command on the left refine the command on the right?
If not, show a counterexample.

```
[] Yes
[] No (Give st and st' below)
      st =
      st' =
```

(ii) Does the command on the right refine the command on the left?
If not, show a counterexample.

```
[] Yes
[] No (Give st and st' below)
      st =
      st' =
```

5.2

```
X := 1
```

```
havoc X
```

(i) Does the command on the left refine the command on the right?
If not, show a counterexample.

```
[] Yes
[] No (Give st and st' below)
       st =
       st' =
```

(ii) Does the command on the right refine the command on the left?
If not, show a counterexample.

```
[] Yes
[] No (Give st and st' below)
       st =
       st' =
```

5.3

```
while true do skip end
```

```
havoc X
```

(i) Does the command on the left refine the command on the right?
If not, show a counterexample.

```
[] Yes
[] No (Give st and st' below)
       st =
       st' =
```

(ii) Does the command on the right refine the command on the left?
If not, show a counterexample.

```
[] Yes
[] No (Give st and st' below)
       st =
       st' =
```

5.4

```
havoc X
```

```
havoc Y
```

(i) Does the command on the left refine the command on the right?
If not, show a counterexample.

```
[] Yes
[] No (Give st and st' below)
       st =
       st' =
```

(ii) Does the command on the right refine the command on the left?
If not, show a counterexample.

```
[] Yes
[] No (Give st and st' below)
       st =
       st' =
```

5.5

```
while X = 0 do havoc X end
```

```
if X = 0 then havoc X end
```

(i) Does the command on the left refine the command on the right?
If not, show a counterexample.

```
[] Yes
[] No (Give st and st' below)
       st =
       st' =
```

(ii) Does the command on the right refine the command on the left?
If not, show a counterexample.

```
[] Yes
[] No (Give st and st' below)
       st =
       st' =
```

$\boxed{6}$ (12 points)

One of the exercises in the Imp chapter introduced a simple, stack-based language and a compiler to it from Imp's arithmetic expressions. We're going to ignore the compiler in this problem and focus on the semantics of the little stack language. In fact, we'll simplify it even further, to just constants and addition:

```
Inductive sinstr : Type :=
| SPush (n : nat)
| SPlus.

Fixpoint sinstr_eval (prog : list sinstr) (stack : list nat) : option (list nat) :=
  match prog with
  | [] => Some stack
  | instr :: prog' =>
    match instr with
    | SPush n => (sinstr_eval prog' (n :: stack))
    | SPlus =>
      match stack with
      | n1 :: n2 :: t =>
        (sinstr_eval prog' (n1 + n2 :: t))
      | _ => None
      end
    end
  end.
```

Your job is to fill in the details of two inductively defined relations that express the same behavior: a big-step evaluation relation and a small-step reduction relation.

$\boxed{6.1}$ Complete this inductively defined relation for the big-step semantics. For example, the following should be provable using the relation you define:

```
sinstr_bstep [SPush 1] [2] [1;2].

sinstr_bstep [SPlus; SPlus] [4;2;1] [7].
```

```
Inductive sinstr_bstep : list sinstr -> list nat -> list nat -> Prop :=
```

$\boxed{6.2}$ Complete this inductively defined relation for the small-step semantics. For example, the following should be provable using the relation you define:

```
sinstr_sstep [SPush 1] [2] [] [1;2].

sinstr_sstep [SPlus; SPlus] [4;2;1] [SPlus] [6;1].
```

```
Inductive sinstr_sstep : list sinstr -> list nat ->
                         list sinstr -> list nat ->
                         Prop :=
```

$\boxed{7}$ **[Advanced Track Only]** (15 points)

Recall the tiny language of constants and addition from the Smallstep chapter.

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

Here is an "indexed" version of the big-step evaluation relation for this language that counts how many addition operations are required to produce the result.

```
Reserved Notation " t '==>' n '//' s" (at level 50).

Inductive eval : tm -> nat -> nat -> Prop :=
  | E_Const : forall n,
      C n ==> n // 0
  | E_Plus : forall t1 t2 n1 n2 i1 i2,
      t1 ==> n1 // i1 ->
      t2 ==> n2 // i2 ->
      P t1 t2 ==> (n1 + n2) // (i1 + i2 + 1)

where " t '==>' n // i" := (eval t n i).
```

For example:

```
Example eg1: (C 6) ==> 6 // 0.

Example eg2: (P (C 6) (C 4)) ==> 10 // 1.

Example eg3: (P (C 6) (P (C 4) (C 3))) ==> 13 // 2.
```

We can also capture the idea of counting addition operations in terms of small-step reduction. We use the same `step` relation as in the Smallstep chapter.

```
Reserved Notation " t '-->' t' " (at level 40).

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').

Lemma nf_same_as_value : forall t,
  normal_form step t <-> value t.
```

Now we enrich the multi-step reduction relation so that it keeps track of a "step counter". The idea is that that the "machine state" at any given moment includes both the expression being evaluated and the value of the step counter. Each step that the machine executes increments the step counter by 1.

```
Reserved Notation " t '@' i '-->*' t' '@' i' "
               (at level 40, t' at level 39, i at level 39, i' at level 39).

Inductive multi_indexed : indexed_relation tm :=
  | multi_refl : forall (x : tm) (i : nat),
                  x @ i -->* x @ i
  | multi_step : forall (x y z : tm) (i j : nat),
                  x --> y ->
                  y @ i -->* z @ j ->
                  x @ i -->* z @ (j+1)

where " t '@' i '-->*' t' '@' i' " := (multi_indexed t i t' i').
```

The `step_eval` lemma can be extended straightforwardly to record the fact that adding one addition step to a multi-step reduction increases step counter by one:

```
Lemma step__eval : forall t t' n i,
     t --> t' ->
     t' ==> n // i ->
     t  ==> n // (i+1).
```

Building on this, we can strengthen the `multistep__eval` theorem to show that the indexed versions of the big- and small-step relations count addition operations in the same way. First, we enrich the `normal_form_of` relation with step indices, as well as add in some standard definitions from the original small step semantics:

```
Definition relation (X : Type) := X -> X -> Prop.
Definition normal_form {X : Type}
              (R : relation X) (t : X) : Prop :=
  ~ exists t', R t t'.

Definition step_normal_form := normal_form step.

Definition normal_form_of (t t' : tm) (i : nat) :=
  (t @ 0 -->* t' @ i) /\ step_normal_form t'.
```

Informally, `normal_form_of t t' i` can be read "term `t` steps to normal form `t'` in `i` steps." More precisely, it says, "If we begin stepping from the term `t` with step counter `0`, then when we reach normal form `t'` the final step counter will be `i`."

Finally, we can argue that big-step evaluation and small-step reduction not only reach the same normal form but do it in the same number of steps. Here is the small-step-to-big-step direction of this claim:

```
Theorem multistep__eval : forall t t' k,
    normal_form_of t t' k ->
    exists m, t' = C m /\ t ==> m // k.
```

Write a careful *informal* proof of this theorem. If your proof uses induction, make sure to state the induction hypothesis *explicitly* at the beginning of each inductive case.