CIS 5000 Midterm 2 12 November 2024

SOLUTIONS

1. **Behavioral Equivalence** (18 points)

   Recall the notion of *program equivalence*, `cequiv`, from the Equiv chapter. Two Imp commands c_1 and c_2 are equivalent if for every starting state `st`, either c_1 and c_2 both diverge or both terminate in the same final state `st'`.

   For the following pairs of Imp programs mark "Equivalent" if the pair of Imp programs are equivalent. If they're not equivalent, mark "Inequivalent" and describe a starting state `st` that exhibits the difference. (Assume that `X`, `Y` and `Z` are distinct variables.)

   (2 points each)

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

   ☒ Equivalent ☐ Inequivalent `st` =

   (b)
   ```
   skip
   ```
   ```
   while true do skip end
   ```

   ☐ Equivalent ☒ Inequivalent `st` = Any starting state

   (c)
   ```
   while X < Y do
     Y := Y + 1;
     X := X - 1;
   end
   ```
   ```
   while true do
     skip
   end
   ```

   ☐ Equivalent ☒ Inequivalent `st` = Any state where X >= Y

   (d)
   ```
   if X > 0 then
     while X <> 0 do
       X := X - 1
       Y := Y - 1
     end
   else skip  end
   ```
   ```
   Y := Y - X;
   X := 0
   ```

   ☒ Equivalent ☐ Inequivalent `st` =

   (e)
   ```
   if X = Y
     then while true do skip end
   else
     X := Y
   end
   ```
   ```
   if X <> Y then
     X := Y
   else
     skip
   end
   ```

   ☐ Equivalent ☒ Inequivalent `st` = Any state where `X = Y`

   (f)
   ```
   while Y < X do
     Y := Y + 1
   end
   ```
   ```
   if Y < X then
     Y := X
   else
     skip
   end
   ```

   ☒ Equivalent ☐ Inequivalent `st` =

Adding Havoc

Now suppose we extend Imp with the `havoc` command from homework. Recall that `havoc x` assigns the variable `x` a natural number value *nondeterministically*.

To achieve this, we add the following large-step rule for `havoc`:

```
                                ---------------------------------  (E_Havoc)
                                 st =[ havoc x ]⇒ (x !→ n ; st)
```

For the following pairs of Imp+`havoc` programs mark "Equivalent" if they are equivalent. If they're not equivalent, mark "Inequivalent" and describe a starting state `st` that exhibits the difference. (Assume that `X`, `Y` and `Z` are distinct variables.)

(2 points each)

(a)

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

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

☐ Equivalent ☒ Inequivalent `st` = when `X=0`

(b)

```
havoc Y; havoc X
```

```
havoc X; havoc Y
```

☒ Equivalent ☐ Inequivalent `st` =

(c)

```
while X > 0 do
   havoc X
end
```

```
X := 0
```

☒ Equivalent ☐ Inequivalent `st` =

2. **Hoare Logic** (30 points total)

Appendix A contains a summary of the standard Hoare Logic rules for Imp.

(a) (10 points) For each Hoare triple below, select *all* the assertions that can fill in the blank to make the triple valid.

i.
```
{{___}} X := 5 {{X = 5}}
```
☒ True ☒ False ☒ X = 5 ☒ X = 7 ☒ Y = 1 ∨ X = 2

ii.
```
{{True}} X := 5; skip {{___}}
```
☒ True ☐ False ☒ False ↠ True ☐ Y = 5 ☒ X = 5

iii.
```
{{X = 0}}
    Y := 0;
    while X = 0 do Y := X + 1 end
{{___}}
```
☒ True ☒ False ☒ X = 0 ☒ Y = 0 ☒ X = 5 ∨ Y = 1

iv.
```
{{___}}
    while X < 5 do X := X + 1 end
{{X = 5}}
```
☐ True ☒ False ☐ X > 5 ☒ X < 5 ☒ X = 5

v.
```
{{___}}
    while X <> Y do
      X := X + 1
      Y := Y + 1
    end
{{X = Y}}
```
☒ True ☒ False ☒ X = 2 ∧ Y = 2 ☒ X = 2 ∨ Y = 2 ☒ Y = 2

(b) (6 points) Which of the following are valid Hoare Triples? (Mark the appropriate box.)

i.
```
{{True}} skip {{False}}
```
☐ Valid ☒ Invalid

ii.
```
{{X = 2}} X := X + 1 {{X = 3}}
```
☒ Valid ☐ Invalid

iii.
```
{{X = 2 ∧ X = 3}} X := 5 {{X = 0}}
```
☒ Valid ☐ Invalid

iv.
```
{{True}} while true do skip end {{False}}
```
☒ Valid ☐ Invalid

v.
```
{{X = 2}} if X = 2 then X := 3 else X := 4 {{X = 3}}
```
☒ Valid ☐ Invalid

vi.
```
{{X = 2}} if X = 3 then X := 3 else X := 4 {{X = 4}}
```
☒ Valid ☐ Invalid

(c) (6 points) Give an example of two (short) assertions, `P` and `Q`, along with a (short) command `c` such that the following specification is satisfied.

```
( {{ P }} c {{ Q }} ↔ {{ P }} c ; c {{ Q }} )     ∧     ~ {{ P }} c {{ P }}.
```

Answer: Many answers will work, one easy solution is:

```
P = X = 0

Q = X = 1

c = X := 1
```

(d) (8 points) The program below is missing the decorations that justify the outer Hoare triple. Fill in the blanks so that the decorations can be used to verify the program. (Intuitively, assuming `X` is even, this program assigns `Y` half of `X`.)

```
{{ X = 2 * n }}
   Y := 0
{{ X = 2 * n ∧ Y = 0 }} →→
{{ X + 2 * Y = 2 * n }} →→
   while (X > 0) do
                  {{ X + 2 * Y = 2 * n ∧ (X > 0) }} →→
                  {{ (X - 2) + 2 * (Y + 1) = 2 * n }}
      X := X - 2
                  {{ X + 2 * (Y + 1) = 2 * n  }};
      Y := Y + 1
                  {{ X + 2 * Y = 2 * n }}
end
{{ X + 2 * Y = 2 * n ∧ ~(X > 0) }} >→ >
{{ Y = n }}
```

3. **Loop Invariants** (10 points total)

Recall that an assertion `P` is a valid loop invariant for `while b do c end` if `{{ P ∧ b }} c {{ P }}` holds.

(a) (5 points) Mark which assertions are valid loop invariants for the following program. For each assertion, also indicate whether proves the assertion implication on the last line. (Choose all that apply)

```
{{ True }}
while X > 0 do
    {{ INV ∧ X > 0 }}
    X := X - 1
    {{ INV }}
end
{{ INV ∧ ∼(X > 0)}} ⇾ {{ X = 0 }}
```

`INV =`	is loop invariant?	`{{ INV ∧ ∼(X > 0)}} ⇾ {{ X = 0 }}` provable?
`True`	☒	☒
`False`	☒	☒
`Y = X * X`	☐	☒
`X ≥ 0`	☒	☒
`X > 0`	☐	☒

(b) (5 points) Mark which assertions are valid loop invariants for the following program. For each assertion, also indicate whether the assertion implication on the last line is provable. (Choose all that apply)

```
{{ True }}
X := 0;
Y := 0;
while X < n do
  {{ INV ∧ X < n }}
  X := X + 1
  Y := Y + (2 * X) - 1;
  {{ INV }}
end
{{ INV ∧  ∼(X < n) }} ⇾ {{ Y = n * n }}
```

`INV =`	is loop invariant?	`{{ INV ∧ ∼(X < n) }} ⇾ {{ Y = n * n }}` provable?
`True`	☒	☐
`Y = X * X`	☒	☐
`Y = X * X ∧ X <= n`	☒	☒
`X <= n`	☒	☐
`(X ≥ Y) ∧ X + Y = n * n`	☐	☐

4. **Variant Imp Semantics** (20 points total)

This problem explores the impact of adding a powerful "search" mechanism to the Imp language. The idea is to add a new command `find X satisfying b`, where `X` is an Imp variable and `b` is a boolean expression that constrains `X`. The idea is that this command assigns `X` a value that satisfies `b` if such a value exists, and fails otherwise.

Example 1: the following command would set `X` to `6` because `6 * 3 = 18`.

```
Y := 18;
find X satisfying (X * 3 = Y)
```

Example 2: On the other hand, the following program would fail (i.e., produce no final state) because `10` is not divisible by `3`:

```
Y := 10;
find X satisfying (X * 3 = Y)
```

Example 3: If more than one value can satisfy the constraint, `find` will choose *nondeterministically*. For example, the following program can set `X` to *any* value bigger than `20`:

```
Y := 10;
find X satisfying (X > 2*Y)
```

(a) **Large Step Semantics** (3 points)

We can define the behavior of `find` by adding a new evaluation rule `(E_Find)` to the usual rules for Imp (see Appendix A). Which of the following rules correctly implements the behavior described above (choose one):

☐

```
                beval st b = true
-------------------------------------------  (E_Find)
st  =[ find X satisfying b ]⇒ (x !→ n; st)
```

☒

```
       beval (x !→ n; st) b = true
-------------------------------------------  (E_Find)
st  =[ find X satisfying b ]⇒ (x !→ n; st)
```

☐

```
                beval st b = true
-------------------------------------------  (E_Find)
      st  =[ find X satisfying b ]⇒ st
```

☐

```
       beval (x !→ n; st) b = true
-------------------------------------------  (E_Find)
  st  =[ find X satisfying b ]⇒ (x !→ n)
```

(b) **Program equivalence** (4 points)

Adding this command affects the notion of Imp program equivalence. Assuming a correct solution to part (a), for the following pairs of Imp programs mark "Equivalent" if the pair of Imp programs are equivalent under the new semantics. If they're not equivalent, mark "Inequivalent" and describe a starting state `st` that exhibits the difference. (Assume that `X`, `Y` and `Z` are distinct variables.)

i.

```
find X satisfying X = 2
```

```
X := 2
```

☒ Equivalent ☐ Inequivalent `st` =

ii.

```
find X satisfying true
```

```
skip
```

☐ Equivalent ☒ Inequivalent `st` = any state (X can become anything)

iii.

```
find X satisfying false
```

```
while true do skip end
```

☒ Equivalent ☐ Inequivalent `st` =

iv.

```
find X satisfying X + Z = Y
```

```
X := 0;
while ~(X + Z = Y) do
  X := X + 1
end
```

☒ Equivalent ☐ Inequivalent `st` =

(c) **Valid Hoare Triples** (5 points)

The notation of validity for Hoare logic remains unchanged, except that now the semantics include the `find` command. Which of the following are valid Hoare triples? (Mark the appropriate box.)

i.

```
{{True}} find X satisfying false {{False}}
```

☒ Valid ☐ Invalid

ii.

```
{{Y = 3}} find X satisfying X + Y = 5 {{X = 2}}
```

☒ Valid ☐ Invalid

iii.

```
{{Y = 3}} find X satisfying Y = 5 {{True}}
```

☒ Valid ☐ Invalid

iv. (Note, the first boolean expression below is logically equivalent to `x = 2` *or* `x = 3`.)

```
{{True}}
 find X satisfying (~(~(X = 2) && ~(X = 3)));
 find Y satisfying (X + 1 = 4))
{{X = 3}}.
```

☒ Valid ☐ Invalid

v.

```
{{True}}
 Y := 10;
 find X satisfying (X > 2*Y)
{{X > 20}}.
```

☒ Valid ☐ Invalid

(d) **Extending the Hoare Logic** (2 points)

With a bit of work, we can define a Hoare logic rule for the `find` command, and we end up with the rule shown below. (Recall that assertions are predicates on states and that the $ notation lets us write a Coq function as an assertion.) *HINT: the form of this rule closely corresponds to the correct answer for part (a) of this question.*

```
{{ P }}
  find X satisfying b
{{ $(fun st' ⇒ ∃ st, P st ∧ ∃ n, st' = (X !→ n; st) ∧
             beval st' b = true) }}.
```

In this rule, the identifier `st` corresponds to which of the following? (choose one)

☒ The state of the program *before* running the `find` command.

☐ A possible state satisfying the constraint `b` that will be the state of the program *after* running the `find` command.

☐ An intermediate state that is used by the `find` command during its "search".

☐ None of the above

(e) **Expressiveness** (2 points)

It turns out that adding `find` makes the language more expressive. This is because, thanks to nondeterminism, the `find` command can "guess" a good answer that will let the program proceed. For example, the following program always terminates with `Y = 3`—even though the `find` command *could* set `X` to `5000` it *can* choose not to, and so this program always terminates.

```
{{ True }}
  find X satisfying (X > 10);
  if X = 5000 then
    while true do skip
  else
    Y = 3
{{ Q }}
```

Given this discussion, what is the *strongest* postcondition `Q` that would be valid for the program above? (Choose one)

☐ `True`

☐ `Y = 3 ∧ ∼(X = 5000)`

☐ `X > 10 ∧ Y = 3`

☒ `X > 10 ∧ ∼(X = 5000) ∧ Y = 3`

☐ `False`

(f) **Implementation** (4 points)

It isn't easy to implement the full power of the `find` command because of the nondeterminism illustrated above. We might try to do it by searching for a good `X`. Consider the following two *inequivalent* programs, where the left is a purported "implementation" of `find`. Because these two commands are not equivalent, they must satisfy different Hoare triples.

```
{{ True }}
  X := 0;
  while ~b do
    X := X + 1
  end
{{ Q }}
```

```
{{ True }}
    find X satisfying b
{{ R }}
```

Assuming that `Q` and `R` are valid, which one will, in general, be stronger? (Choose one)

☒ `Q` ⇒ `R` (i.e., `Q` is stronger) ☐ `R` ⇒ `Q` (i.e., `R` is stronger)

Why? (briefly explain):

Answer: Postcondition `Q` *is stronger because, more than just finding an* `X` *satisfying* `b`*, the left-hand program finds the unique, smallest such value. That is a much more specific (i.e. stronger) assertion about the final state of the program. Another way of saying the same thing is that the left program refines the right program.*

Grading scheme: 1 point for the correct choice. 3 points for an explanation along the lines of the above, with partial credit given.

5. **Small Step Semantics** (12 points total)

   Appendix B contains the definitions of the toy language of constants (`C n`) and addition (`P t1 t2`) from `Smallstep.v`, along with the definitions of both the large and small-step semantics from the homework.

   (a) (2 points) Suppose we replace the rule `ST_Plus2` with the version below that drops the `value` premise:

```
        t2 ⟶ t2'
-------------------- (ST_Plus2')
P v1 t2 ⟶ P v1 t2'
```

   Which of the following properties still hold? (Mark all that hold)

   ☒ `normalizing`

   ☒ `value_is_nf`

   ☒ `nf_is_value`

   ☐ `deterministic`

   (b) (2 points) Suppose we instead replace the rule `ST_plus1` with the version below:

```
-------------------- (ST_Plus1')
P t1 t2 ⟶ P t2 t1
```

   Which of the following properties still hold? (Mark all that hold)

   ☐ `normalizing`

   ☒ `value_is_nf`

   ☒ `nf_is_value`

   ☐ `deterministic`

   (c) (2 points) Suppose that we instead change the definition of `value` so that only non-zero constants are considered values, i.e. `value (C (S n))` and `not (value (C 0))`. Which of the properties still hold? (Mark all that hold)

   ☒ `normalizing`

   ☒ `value_is_nf`

   ☐ `nf_is_value`

   ☒ `deterministic`

Adding Multiplication (6 points)

Now suppose that (instead of any of the changes above) we want to extend this language with a multiplication operation `M t t` that *multiplies* its arguments after they are reduced to constants. The notion of value remains unchanged. We first extend the grammar and add the new large step rule as shown:

```
t ::= C n                    values  v := C n
    | P t t
    | M t t

          t1 ⇒ n1
          t2 ⇒ n2
       ------------------ (E_Plus)
       M t1 t2 ⇒ n1 * n2
```

In the space below, write the rules necessary to define the small step operational semantics for `M t1 t2` that agree with the large-step. Define them such that (1) the rules are *deterministic* and *normalizing* and (2) `t2` is evaluated *before* `t1` for the term `M t1 t2`. (Assume that the rest of the rules are exactly as shown in Appendix B.)

```
-------------------------------  (ST_MultConstConst)
M (C n1) (C n2) ⟶ C (n1 * n2)

          t2 ⟶ t2'
       -------------------- (ST_Mult1)
       M t1 t2 ⟶ M t1 t2'

          value v2
          t1 ⟶ t1'
       -------------------- (ST_Mult2)
       M t1 v2 ⟶ M t1' v2
```

CIS 5000 2024 Midterm 2 Appendices

(Do not write answers in the appendices. They will not be graded)

Appendix A: Imp Semantics and Hoare Logic Rules

Imp Large Step Semantics

```
                          -----------------                                (E_Skip)
                          st =[ skip ]⇒ st

                           aeval st a = n
                      ------------------------------                       (E_Asgn)
                      st =[ x := a ]⇒ (x !→ n ; st)

                           st  =[ c1 ]⇒ st'
                           st' =[ c2 ]⇒ st''
                          ---------------------                             (E_Seq)
                          st =[ c1;c2 ]⇒ st''

                           beval st b = true
                            st =[ c1 ]⇒ st'
                    ---------------------------------------                 (E_IfTrue)
                    st =[ if b then c1 else c2 end ]⇒ st'

                          beval st b = false
                            st =[ c2 ]⇒ st'
                    ---------------------------------------                 (E_IfFalse)
                    st =[ if b then c1 else c2 end ]⇒ st'

                          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''
```

Imp Hoare Logic Rules

```
                  --------------------------- (hoare_asgn)
                  {{Q [X ⊢> a]}} X:=a {{Q}}

                  --------------------  (hoare_skip)
                  {{ P }} skip {{ P }}

                    {{ P }} c1 {{ Q }}
                    {{ Q }} c2 {{ R }}
                   ---------------------  (hoare_seq)
                   {{ P }} c1;c2 {{ R }}

                   {{P ∧    b}} c1 {{Q}}
                   {{P ∧ ~ b}} c2 {{Q}}
          ------------------------------------  (hoare_if)
          {{P}} if b then c1 else c2 end {{Q}}

                  {{P ∧ b}} c {{P}}
            ---------------------------------  (hoare_while)
            {{P}} while b do c end {{P ∧ ~ b}}

                  {{P'}} c {{Q'}}
                     P →ₚ P'
                     Q' →ₚ Q
             --------------------------   (hoare_consequence)
                  {{P}} c {{Q}}
```

Appendix B: Smallstep Toy Language Definitions

```
    t ::= C n                    values  v := C n
        | P t t
```

```
Inductive tm : Type :=
  | C : nat → tm (* Constant *)
  | P : tm → tm → tm. (* Plus *)

Inductive value : tm → Prop :=
  | v_const : ∀ n, value (C n).
```

Large Step Operational Semantics:

```
                 -------- (E_Const)
                 C n ⇒ n


                 t1 ⇒ n1
                 t2 ⇒ n2
           ------------------ (E_Plus)
           P t1 t2 ⇒ n1 + n2
```

Small Step Operational Semantics:

```
  ------------------------------  (ST_PlusConstConst)
  P (C n1) (C n2) ⟶ C (n1 + n2)

               t1 ⟶ t1'
          -------------------- (ST_Plus1)
          P t1 t2 ⟶ P t1' t2

               value v1
               t2 ⟶ t2'
          -------------------- (ST_Plus2)
          P v1 t2 ⟶ P v1 t2'
```

Definitions

Given `value` and `step` relations defined above, we have the following additional definitions.

```
Definition normal_form t : Prop :=
  ~∃ t', step t t'.

Definition deterministic :=
  ∀ t t1 t2, (t ⟶ t1) → (t ⟶ t2) → t1 = t2.

Lemma normalizing : ∀ t,
    ∃ t', (multi step) t t' ∧ normal_form t'.

Lemma value_is_nf : ∀ v,
    value v → normal_form v.

Lemma nf_is_value : ∀ nf,
    normal_form nf → value nf.

Inductive multi {X : Type} (R : relation X) : relation X :=
  | multi_refl : ∀ (x : X), multi R x x
  | multi_step : ∀ (x y z : X),
                   R x y →
                   multi R y z →
                   multi R x z.
```