CIS 5000: Software Foundations **Midterm II**

November 16, 2023

Solutions

1 [Standard Track Only] T/F Questions (14 points)

(a) This theorem can be proved in Coq without any axioms:

```
forall (A : Type) (m : partial_map A) x v1 v2,
  (x |-> v2 ; x |-> v1 ; m) = (x |-> v2 ; m)
```

True [✓ False]

(b) If two Imp commands `c1` and `c2` are equivalent (that is, `st =[c1]=> st'` iff `st =[c2]=> st'`, for all `st` and `st'`), then they also validate the same Hoare triples (that is, `{{P}} c1 {{Q}}` iff `{{P}} c2 {{Q}}`, for all `P` and `Q`).

[✓ True] False

(c) Conversely, if two Imp commands validate the same Hoare triples, then they are equivalent.

[✓ True] False

(d) If `c1` is equivalent to `c1;c2`, then `c1;c2` is equivalent to `c1;c2;c2`.

[✓ True] False

(e) For all `c` and `Q`, the Hoare triple `{{True}} c {{Q}}` is valid.

True [✓ False]

(f) For all `P` and `Q`, the Hoare triple `{{P}} while true do skip {{Q}}` is valid.

[✓ True] False

(g) If `P` and `Q` are assertions, we say that "`P` implies `Q`" if there exists a state `st` such that `P st` implies `Q st`.

True [✓ False]

(h) The Hoare triple `{{X = n}} X := 2 * X {{X := 2 * X}}` is valid.

True [✓ False]

(i) The small-step and big-step semantics of a language are related in that a big step is defined to be zero or more small steps. That is, the big-step relation is defined to be the reflexive, transitive closure of the small-step relation.

True [✓ False]

(j) If the small-step relation for a language satisfies strong progress, then the language must also be deterministic.

True [✓ False]

(k) Conversely, if the small-step relation is deterministic, it must also satisfy strong progress.

True [✓ False]

(l) Suppose we have a language where *no* term is well-typed. Then, regardless of how the step relation is defined, progress and preservation will hold.

[✓ True] False

(m) Suppose we have a language where *every* term is well-typed. Then, regardless of how the step relation is defined, progress and preservation will hold.

True [✓ False]

(n) Suppose we have a language for which preservation holds. Then adding new typing rules will never cause preservation to fail.

True [✓ False]

$\boxed{2}$ **Program Equivalence** (14 points)

Recall that two Imp commands c_1 and c_2 are *equivalent* (in Coq, `cequiv`) if, for every starting state `st`, either c_1 and c_2 both diverge or else they both terminate in the same final state `st'`. In this problem we will explore some variations on this definition.

(a) For the following pairs of Imp programs, check the appropriate box to indicate whether the two programs will both diverge or both terminate in the same final state from *every* starting state (i.e., they are equivalent), from *some but not all* starting states, or from *no* starting states at all.

If you choose the *"Some but not all starting states"* option, then you should also provide an example starting state under which the two programs will behave the same (both diverge or both terminate in the same final state) and a starting state where they will behave differently. For example:

```
skip
```

```
X := 0
```

☐ All starting states [✓ Some starting states] ☐ No starting states

If you choose "Some starting states"...

These behave the same on state: `[ X |-> 0 ]` (or `[]`)
These behave differently on state: `[ X |-> 1 ]`

You do the rest...

i.

```
X := 2
```

```
Y := 1;
X := Y;
X := X + 1
```

☐ All starting states [✓ Some starting states] ☐ No starting states

If you choose "Some starting states"...
These behave the same on state: `[ Y |-> 1 ]`
These behave differently on state: `[ Y |-> 0 ]`

ii.

```
X := 2
Y := 1
while X > Y do
  skip
end
```

```
X := 2
Y := 1
while X > Y do
  X := X - 1
end
```

☐ All starting states ☐ Some starting states [✓ No starting states]

If you choose "Some starting states"...

These behave the same on state:

These behave differently on state:

iii.

```
while X > Y do
  X := X + 1
end
```

```
while Y >= X do
  Y := Y + 1
end
```

☐ All starting states ☐ Some starting states [✓ No starting states]

If you choose "Some starting states"...

These behave the same on state:

These behave differently on state:

iv.

```
if X = Y then
  while true do
    skip
  end
else
  skip
end
```

```
while X = Y do
  X := X + 2;
  Y := Y + 2
end
```

[✓ All starting states] ☐ Some starting states ☐ No starting states

If you choose "Some starting states"...

These behave the same on state:

These behave differently on state:

v.

```
X := 1;
while X <> Y do
  X := X + 1
end
```

```
X := Y
```

☐ All starting states ☒ Some starting states ☐ No starting states

If you choose "Some starting states"...

These behave the same on state:	`[ X |-> 5; Y |-> 5 ]`
These behave differently on state:	`[ X |-> 0; Y |-> 0 ]`

(b) The left-hand box in each problem below contains a complete Imp program, while the right-hand box contains a program with a missing expression or command. Choose the code snippet below the boxes that, when used to fill in the blank, will make the programs behaviorally equivalent. If none of the options will make the programs behaviorally equivalent, choose *"None of these"*.

i.

```
skip
```

```
while __________ do
  skip
end
```

☐ `X <> 0` ☐ `true` ☒ `false` ☐ None of these

ii.

```
Z := X;
X := Y;
Y := Z
```

```
Z := Y;
Y := X;
X := Z;
__________
```

☐ `Y := Z` ☒ `Z := Y` ☐ `Y := X` ☐ None of these

iii.

```
while X < Y do
  X := X + 1
end
```

```
while X < Y do
  while _________ do
    X := X + 1
  end
end
```

☐ `true` ☒ `X < Y` ☐ `Y < X` ☐ None of these

iv.

```
Y := Y + X
```

```
Z := X
while Z <> 0 do
  Z := Z - 1;
  __________
end
```

☐ `Y := Y + 1` ☐ `Y := Y + X` ☐ `Y := X + Z` ☒ None of these

$\boxed{3}$ **Loop Invariants** (15 points)

In this problem, we are 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`, a loop invariant `P`, and a final postcondition `F`:

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

To be completely correct, such a partially decorated program must satisfy three conditions:

(a) the loop invariant `P` must be *established* by the initialization steps—that is, the Hoare triple `{{I}} c1 {{P}}` must be valid;

(b) the loop invariant can must be *preserved* by the loop body—that is, the triple `{{P/\b}} c2 {{P}}` must be valid; and

(c) the loop invariant together with the fact that the loop guard is false must *imply* the desired *final* condition—that is, `P/\~b` must imply `F`.

Moreover, we may want to know whether the loop is *guaranteed to terminate* when started in any state satisfying `P`.

Below, we give several Imp programs with initial preconditions and final postconditions. For each one, we also give some candidate loop invariants.

For each candidate invariant, check one or more of the boxes to indicate whether `P` is *established* by `I`, whether it is *preserved* by the loop body, whether it *implies* the *final* postcondition, and whether it *guarantees termination* of the loop.

$\boxed{3.1}$

```
{{ True }}
  skip;
{{ P }}
  while X >= Y do
    {{ P /\ X >= Y }}
      Z := X;
      X := Y;
      Y := Z
    {{ P }}
  end
{{ P /\ ~(X >= Y) }} ->>
{{ X <= Y }}
```

`P = True`

☒ established ☒ preserved ☒ implies final ☐ guarantees termination

`P = False`

☐ established ☒ preserved ☒ implies final ☒ guarantees termination

`P = Y < X`

☐ established ☐ preserved ☒ implies final ☒ guarantees termination

$\boxed{3.2}$

```
  {{True}}
    X := n;
    Z := 0;
  {{ P }}
    while 1 <= X do
      {{ P /\ 1 <= X }}
        Z := Z + 1
        X := X - 2
      {{ P }}
    end
  {{ P /\ ~(1 <= X) }} ->>
  {{ 2 * Z + 1 = n \/ 2 * Z = n}}
```

`P = X = n`

☒ established ☐ preserved ☐ implies final ☒ guarantees termination

`P = Z = n`

☐ established ☐ preserved ☐ implies final ☒ guarantees termination

`P = 2 * Z = n - X`

☒ established ☐ preserved ☒ implies final ☒ guarantees termination

$\boxed{3.3}$

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

`P  =  X = 0 /\ Y = 1`

☐ established ☒ preserved ☒ implies final ☒ guarantees termination

`P  =  Z + X = n`

☒ established ☒ preserved ☐ implies final ☒ guarantees termination

`P  =  Y <= X`

☐ established ☒ preserved ☐ implies final ☒ guarantees termination

$\boxed{4}$ **Variations on Validity** (12 points)

Recall the standard definition of a valid Hoare triple `{{P}} c {{Q}}`:

```
forall st st', st =[ c ]=> st' -> P st -> Q st'.
```

We'll be exploring the consequences of some alternative definitions.

Let's say that a triple is *originally valid* if it is satisfies the original definition, and that a triple is *alternatively valid* if it satisfies the alternative definition; likewise for *originally invalid* and *alternatively invalid*. You will be asked to provide examples of triples `{{P}} c {{Q}}` that demonstrate some combination of these properties, or to say that there are no such triples.

(a) Alternative definition:

```
forall st st', st =[ c ]=> st' -> P st /\ Q st'.
```

Provide an example of a triple that is originally valid and alternatively valid.

☐ There are no such triples.

☒ Here's one:
```
P = {{True}}
c = skip
Q = {{True}}
```

Provide an example of a triple that is originally valid and alternatively invalid.

☐ There are no such triples.

☒ Here's one:
```
P = {{False}}
c = skip
Q = {{True}}
```

Provide an example of a triple that is originally invalid and alternatively invalid.

☐ There are no such triples.

☒ Here's one:
```
P = {{True}}
c = skip
Q = {{False}}
```

Provide an example of a triple that is originally invalid and alternatively valid.

☒ There are no such triples.

☐ Here's one:
```
P =
c =
Q =
```

(b) Alternative definition:

```
forall st, P st -> exists st', Q st'.
```

Provide an example of a triple that is originally valid and alternatively valid.

☐ There are no such triples.

☒ Here's one:
```
P =  {{True}}
c =  skip
Q =  {{True}}
```

Provide an example of a triple that is originally valid and alternatively invalid.

☐ There are no such triples.

☒ Here's one:
```
P =  {{True}}
c =  while true do skip end
Q =  {{False}}
```

Provide an example of a triple that is originally invalid and alternatively invalid.

☐ There are no such triples.

☒ Here's one:
```
P =  {{True}}
c =  skip
Q =  {{False}}
```

Provide an example of a triple that is originally invalid and alternatively valid.

☐ There are no such triples.

☒ Here's one:
```
P =  {{X = 0}}
c =  X := 1
Q =  {{X = 0}}
```

$\boxed{5}$ **A Typed, Small-Step Stack Machine** (15 points)

In this problem your task will be to design a typing relation for the world's simplest stack machine.

The machine itself consists of a straight-line program—a list of instructions—plus a stack that can hold both numeric and boolean values.

```
Inductive stack_element :=
| B : bool -> stack_element
| N  : nat -> stack_element.

Definition stack := list stack_element.

Inductive instr :=
| PUSH : nat -> instr
| ADD : instr
| AND : instr.

Definition prog := list instr.
```

A single step of the machine executes a single instruction, taking a starting stack to an ending stack.

```
Reserved Notation "i '/' s '-->' s'" (at level 40).

Inductive singlestep : instr -> stack -> stack -> Prop :=
| ST_push : forall s n,
    PUSH n / s --> (N n :: s)
| ST_add : forall s n1 n2,
    ADD / (N n1 :: N n2 :: s) --> (N (n1 + n2) :: s)
| ST_and : forall s b1 b2,
    AND / (B b1 :: B b2 :: s) --> (B (b1 && b2) :: s)

where "i / s '-->' s'" := (singlestep i s s').
```

$\boxed{5.1}$ This machine can get stuck in some situations. That is, there exist instructions `i` and stacks `s` such that `i / s --> s'` does *not* hold for any `s'`.

Give an example of such an `i` and `s`.

Answer: This can happen when there are not enough operands on the stack, or the top stack elements do not have the right types. E.g.:

```
Example stepeg0 :
  ~ exists s',
     ADD / [N 4] --> s'.

Example stepeg1 :
  ~ exists s',
      ADD / [N 4; B false] --> s'.
```

Are there any instructions `i` that *cannot* get stuck, no matter what `s` they are executed with?

[✓ Yes] ☐ No

(Namely `PUSH` *instructions.)*

$\boxed{5.2}$ To typecheck programs for this machine, we begin by assigning types to individual stack elements and to whole stacks.

```
Inductive ty := BOOL | NAT.

Definition stack_ty := list ty.

Inductive elt_has_type : stack_element -> ty -> Prop :=
| ET_NAT : forall n, elt_has_type (N n) NAT
| ET_BOOL : forall b, elt_has_type (B b) BOOL.

Reserved Notation "'|-' s '\in*' sty" (at level 40).

Inductive stack_has_type : stack -> stack_ty -> Prop :=
| SHT_nil : |- [] \in* []
| SHT_cons : forall s sty e T,
    |- s \in* sty ->
    elt_has_type e T ->
    |- (e::s) \in* (T::sty)

where "'|-' s '\in*' sty" := (stack_has_type s sty).
```

The typing relation for instructions relates an instruction and *two* types, one describing the stack before the instruction executes and one for the state after. For example:

```
Example eg0 :
  |- PUSH 4 \in [] --> [NAT].

Example eg1 :
  |- PUSH 4 \in [NAT] --> [NAT; NAT].
```

We also, of course, want the instruction typing relation and the `singlestep` relation to fit together in the expected way:

```
Lemma singlestep_preserves_typing : forall i s sty s' sty',
    |- i \in sty --> sty' ->
    |- s \in* sty ->
    i / s --> s' ->
    |- s' \in* sty'.
```

Complete the definition of the instruction typing relation on the next page...

```
Reserved Notation "'|-' i '\in' st --> st'" (at level 40).

Inductive has_type : instr -> stack_ty -> stack_ty -> Prop :=
```

Answer:

```
| T_push : forall sty n,
    |- PUSH n \in sty --> (NAT :: sty)
| T_add : forall sty,
    |- ADD \in (NAT :: NAT :: sty) --> (NAT :: sty)
| T_and : forall sty,
    |- AND \in (BOOL :: BOOL :: sty) --> (BOOL :: sty)

  where "|- i \in sty '-->' sty'" := (has_type i sty sty').
```

5.3 The one-instruction `singlestep` relation can be lifted to a multi-step reduction relation that executes zero or more instructions, threading the ending stack from each into the starting stack for the next.

```
Reserved Notation "p '/' s -->* s'" (at level 40).

Inductive multistep : prog -> stack -> stack -> Prop :=
  | multi_refl : forall (s : stack), [] / s -->* s
  | multi_singlestep : forall i p (s s_mid s' : stack),
      i / s --> s_mid ->
      p / s_mid -->* s' ->
      (i::p) / s -->* s'

where "p '/' s -->* s'" := (multistep p s s').
```

Use the instruction typing relation above to define a similar relation describing how executing a whole program changes the shape of the stack.

```
Example eg2 :
  |- [PUSH 4; PUSH 6]
     \in [BOOL] -->* [NAT; NAT; BOOL].
```

Again, make sure your definition of multi-step typing fits with multi-step reduction:

```
Lemma singlestep_preserves_typing : forall i s sty s' sty',
    |- i \in sty --> sty' ->
    |- s \in* sty ->
    i / s --> s' ->
    |- s' \in* sty'.
```

Complete the definition on the next page...

```
Reserved Notation "'|-' p '\in' st '-->*' st'" (at level 40).

Inductive prog_has_type : prog -> stack_ty -> stack_ty -> Prop :=
```

Answer:

```
| prog_nil : forall sty, |- [] \in sty -->* sty
| prog_cons : forall i p sty sty_mid sty',
    |- i \in sty --> sty_mid ->
    |- p \in sty_mid -->* sty' ->
    |- (i :: p) \in sty -->* sty'

where "'|-' p '\in' st '-->*' st'" := (prog_has_type p st st').
```

$\boxed{6}$ **[Advanced Track Only]** (14 points)

Recall the Hoare Logic rule for `while` loops:

```
Theorem hoare_while : forall P (b:bexp) c,
  {{P /\ b}} c {{P}} ->
  {{P}} while b do c end {{P /\ ~b}}.
```

Write a careful informal proof of this theorem. If your proof uses induction, be explicit about the exact form of the induction hypothesis.

Proof: Recall the definition of valid Hoare triples: `{{Q}} d {{R}}` means that, for any state `st` satisfying `Q`, if `st =[d]=> st'`, then `st'` satisfies `R`.

Suppose we are given a derivation of `st =[while b do c end]=> st'`. We argue, by induction on this derivation, that if `st` satisfies `P`, then `st'` satisfies `P /\ ~b`. There are two cases to consider (the others are contradictory):

(a) `st =[while b do c end]=> st'` by rule `E_WhileFalse`, with `beval st b = false` and `st = st'`. The fact that `st` satisfies `P /\ ~b` is immediate.

(b) `st =[while b do c end]=> st'` by rule `E_WhileTrue`, with

- `beval st b = true`
- `st =[c]=> st''`
- `st'' =[while b do c end]=> st'`
- if `st''` satisfies `P`, then `st'` satisfies `P /\ ~B` (the IH).

The first and second of these, together with the hypothesis `{{P /\ b}} c {{P}}`, imply that `st''` satisfies `P`. The conclusion then follows from the IH.

For Reference

Imp Big-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''
```

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'
                   Q' ->> Q
               ----------------            (hoare_consequence)
                {{P}} c {{Q}}
```