CIS 5000: Software Foundations **Midterm II**

November 6, 2025

Solutions

1 **Equivalence-When P** (14 points total)

Programs `c` and `d` are said to be `equivalent-when P` if, whenever they are started in a state satisfying `P`, either both diverge or both terminate in the same state.

For example, the following two programs are `equivalent-when X = 0`:

```
if X > 0 then
  Y := 1
else
  Y := 0
```

```
Y := 0
```

but they are *not* `equivalent-when X < 5`.

For each of the following questions (1.1–1.6), *mark all that apply.*

1.1 (6 points) Consider the following two programs:

```
if X = 0 then
  if Y > 0 then
    Z := Y
  else
    Z := 0
else
  if Y = 0 then
    Z := X
  else
    Z := 0
```

```
if Y > 0 then
  Z := Y
else
  Z := X
```

For which assertions `P` in the following list are these two programs `equivalent-when P`?

- ☒ `X = 0`
- ☐ `X > 0`
- ☒ `Y = 0`
- ☐ `Y > 0`
- ☐ `X = Y`
- ☐ None of these

1.2 (3 points) If `c` and `d` are `equivalent-when P` and `d` and `e` are `equivalent-when Q`, then `c` and `e` are guaranteed to be...

- ☐ `equivalent-when P`
- ☐ `equivalent-when Q`
- ☒ `equivalent-when P /\ Q`
- ☐ `equivalent-when P \/ Q`
- ☒ `equivalent-when False`
- ☐ `equivalent-when True`
- ☐ None of these

1.3 (2 points) If `c` and `d` are `equivalent-when P` and `P` implies `Q`, then `c` and `d` are guaranteed to be...

- ☐ `equivalent-when Q`
- ☒ `equivalent-when P /\ Q`
- ☐ `equivalent-when P \/ Q`
- ☐ None of these

1.4 (3 points) If `c` and `d` are `equivalent-when b` and `{{b}} c {{Q}}` is a valid Hoare triple, then...

- ☐ `c` and `d` are `equivalent-when Q`
- ☒ `{{b}} d {{Q}}` is a valid Hoare triple
- ☒ `(if b then d else c end)` and `c` are `equivalent-when True`
- ☒ `{{True}} if b then d else c end {{Q}}` is a valid Hoare triple
- ☐ None of these

The second, third, and fourth answers are all valid. 2 is straightforward: `c` *and* `d` *will end in the same state, so if* `c` *satisfies* `Q` *then so does* `d`*. 3 is valid because if* `b` *is true then we are in a state satisfying* `b`*, and if we aren't satisfying* `b` *we take the false branch and execute the same program as* `c`*. 4 is true by a similar argument.*

2 **Loop Invariants** (18 points total)

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 *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 correct, 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`.

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

2.1 (8 points) The greatest common divisor (GCD) of two numbers m and n, written `gcd` m n, is the largest number that evenly divides both m and n. The following `Imp` implementation of the *Euclidean algorithm* computes the GCD of two numbers. It has been annotated with preconditions, postconditions, and several candidate loop invariants.

For each candidate, check all the appropriate boxes to indicate whether `P` is (a) *established* by the initialization steps and the initial precondition, (b) *preserved* by the loop body, and (c) *implies* the *final* postcondition, and whether it (d) *guarantees termination* of the loop.

```
{{ X = n, Y = m }}
  skip;
{{ P }}
  while X <> Y do
    {{ P /\ X <> Y }}
    if X > Y then
      X := X - Y
    else
      Y := Y - X
    end
    {{ P }}
  end
{{ P /\ ~(X <> Y) }} ->>
{{ X = #gcd n m }}
```

`P is... True`

☒ established ☒ preserved ☐ implies final ☒ guarantees termination

`P is... False`

☐ established ☒ preserved ☒ implies final ☒ guarantees termination

`P is... #gcd X Y = #gcd n m`

☒ established ☒ preserved ☒ implies final ☒ guarantees termination

During the exam we realized that allowing m or n to be zero would be confusing because then gcd n m would arguably be ill defined. So we took the correct answer to be "guarantees termination" in all cases.

2.2 (5 points) Give a valid loop invariant `P` for the following program:

```
{{ X = m }}
  Y := 0
{{ P }}
  while X <> 0 do
    {{ P /\ X <> 0}}
      X := X - 1;
      Y := Y + 1
    {{ P }}
  end
{{ P /\ X = 0 }} ->>
{{ Y = m }}
```

Answer:

```
P is...   Y + X = m
```

Does your loop invariant, in addition to satisfying the other three conditions, also guarantee termination of the loop?

☐ Yes ☐ No

2.3 (5 points) Give a valid loop invariant `P` for the following program:

```
{{ X = m /\ Y = n }}
  Z := 1
{{ P }}
  while Y <> 0 do
    {{ P /\ Y <> 0}}
      Z := X * Z;
      Y := Y - 1
    {{ P }}
  end
{{ P /\ Y = 0 }} ->>
{{ Z = #exp m n }}
```

Answer:

```
P is...   Z = #exp m (n - Y) /\ X = m
```

Does your loop invariant, in addition to satisfying the other three conditions, also guarantee termination of the loop?

☐ Yes ☐ No

3 Imp with Conditional Atomic Blocks (17 points total)

In this problem, we consider an extension of Imp with *conditional atomic blocks* of the form `attempt c andcheck b`.

The intended semantics are as follows: (1) run the inner command `c`, (2) check the boolean condition `b`, (3) if `b` holds, proceed; otherwise, roll back to the state at the beginning of the `attempt` block and proceed.

Consider the following example:

```
Definition command1 :=
  <{
    attempt
        X := 1
    andcheck X = 0
  }>.
```

Beginning with a state where `X` is `0`, we execute the assignment `X := 1` within the `attempt` block, then check that the condition `X = 0` holds. It doesn't, so we roll back to the original state.

Now we can make statements like:

```
(X !-> 0) =[ command1 ]=> (X !-> 0).
```

This says that starting `command1` in a state with `X` set to `0` results in the same state, since the assignment was rolled back.

One more example:

```
Definition command2 :=
<{
  attempt
      Y := 2 * X
  andcheck Y = 0
}>.
```

Unlike the previous example, the body of this `attempt` *can* succeed, depending on the starting state:

```
(X !-> 0) =[ command2 ]=> (Y !-> 0; X !-> 0).
```

In this example, starting from a state with `X` set to `0`, we execute the assignment `Y := 2 * X`, which sets `Y` to `0`. The condition `Y = 0` holds, so we don't roll back. However, any execution where `X` is positive will *not* succeed:

```
(X !-> 1) =[ command2 ]=> (X !-> 1).
```

3.1 (3 points) Fill in the final state such that the statement below the program holds.

```
Definition command_ex1 :=
  <{  attempt
        X := 1;
        attempt
          X := 3
        andcheck X <= 2;
        X := X + 1
      andcheck X > 1
  }>.

(X !-> 0) =[ command_ex1 ]=> (X !-> 2)
```

3.2 (3 points) Fill in the final state such that the statement below the program holds.

```
Definition command_ex2 :=
  <{  attempt
        X := X + 1;
        Y := X + 2;
        attempt
          X := 2 * Y
        andcheck false
      andcheck X = 6
  }>.

(X !-> 0; Y !-> 0) =[ command_ex2 ]=> (X !-> 0; Y !-> 0)
```

3.3 (8 points) Complete the big-step evaluation rules for this new construct:

```
Inductive ceval : com -> state -> state -> Prop :=
  (* Other rules as before *)
```

Answer:

```
  | E_AttemptTrue: forall st st' c b,
      st =[ c ]=> st' ->
      beval st' b = true ->
      st =[ attempt c andcheck b ]=> st'

  | E_AttemptFalse: forall st st' c b,
      st =[ c ]=> st' ->
      beval st' b = false ->
      st =[ attempt c andcheck b ]=> st


where "st =[ c ]=> st'" := (ceval c st st').
```

3.4 (3 points) Let's write a Hoare rule for the `attempt c andcheck b` extension. What postcondition should we put in the blank to best capture the construct's behavior?

```
                          {{P}} c {{Q}}
-----------------------------------------------------
{{P}} do c andcheck b {{ _______ }}
```

☒ `P \/ (b /\ Q)`

☐ `(~b /\ P) \/ (b /\ Q)`

☐ `(~b /\ P) \/ Q`

☐ `b /\ (P \/ Q)`

Explanation: `P \/ (b /\ Q)` is a correct postcondition because either:

- The condition `b` failed after running `c`, in which case the state was restored to its original value. Because the precondition `P` had to be true then, it must be true again. Note that `b` may not have been true then, and so may not be true now.
- The condition `b` succeeded right after running `c`. The state was not modified further after checking `b`, so both `b` and `Q` have to be true.

Postconditions `(~b /\ P) \/ (b /\ Q)` would not work, as demonstrated by

```
{{ X = 0 }}
do
  {{ X = 0 }}
  X := 1
  {{ X = 1 }}
andcheck X = 0
{{ (~(X = 0) /\ X =0) \/ (X = 0 /\ X = 1) }}
```

An initial state `X !-> 0` would result in a final state `X !-> 0`, which does not satify the postcondition.

Postconditions `(~b /\ P) \/ Q` would not work, as demonstrated by

```
{{ X = 0 }}
do
  {{ X = 0 }}
  X := 1
  {{ X = 1 }}
andcheck X = 0
{{ (~(X = 0) /\ X =0) \/ (X = 1) }}
```

An initial state `X !-> 0` would result in a final state `X !-> 0`, which does not satify the postcondition.

Postconditions `b /\ (P \/ Q)` would not work, as demonstrated by

```
{{ X = 0 }}
do
  {{ X = 0 }}
  X := 1
  {{ X = 1 }}
andcheck X = 2
```

```
{{ (X = 2) /\ (X = 0 \/ X = 1) }}
```

An initial state `X !-> 0` would result in a final state `X !-> 0`, which does not satify the postcondition.

4 Imp + Break (16 points total)

In this problem we will consider a variant of Imp extended with a `break` statement that, when encountered in a `while` loop, immediately terminates execution of the loop. For example, this program will terminate with `X` set to 1:

```
while X < 100 do
    X := 1;
    break;
    X := 0
end
```

The syntax of our language is as follows. Relevant `Notation` is defined in the appendix.

```
Inductive com : Type :=
  | CSkip
  | CBreak                              (* <------ new *)
  | CAsgn (x : string) (a : aexp)
  | CSeq (c1 c2 : com)
  | CIf (b : bexp) (c1 c2 : com)
  | CWhile (b : bexp) (c : com).
```

To write down the evaluation relation, we also need to distinguish between standard termination and termination via `break`. We accomplish this with the following data type:

```
Inductive result : Type :=
  | SContinue
  | SBreak.
```

Intuitively, `st =[ c ]=> st' / s` means that, if `c` is started in state `st`, then it terminates in state `st'` and either signals that the innermost surrounding loop should exit immediately (when `s = SBreak`) or that execution should continue normally (when `s = SContinue`). Formally, for any command and initial state:

- If the command is `skip`, we terminate normally with the state unchanged.
- If the command is `break`, we terminate in a `break` with the state unchanged.
- If the command is an assignment `x := a`, we evaluate `a` to a value `n`, terminate normally, and update the state to map `x` to `n`.
- If the command is a sequence `c1; c2`, we evaluate `c1`.
  - If `c1` terminates normally, we then evaluate `c2` from the resulting state.
  - If `c1` terminates with a `break`, we terminate with a `break` in the state resulting from `c1`.
- If the command is a conditional of the form `if b then c1 else c2 end`, we evaluate `b`. If `b` is `true`, we execute `c1`, otherwise `c2`.
- If the command is a loop of the form `while b do c end`, we evaluate `b`.
  - If `b` evaluates to `false`, we terminate normally with the state unchanged.
  - If `b` evaluates to `true`, we evaluate `c`. If `c` terminates normally, we re-evaluate the loop in `c`'s resulting state. If `c breaks`, the loop terminates normally in `c`'s resulting state.

4.1 (2 points) Does the following program terminate? If so, in what state?

```
X := 0;
while true do
    while true do
      break
    end;
    if X < 10 then
      X := X + 1
    else
      break
    end
end
```

☐ Does not terminate

☒ Terminates with

```
X  =  10
```

4.2 (2 points) Does the following program terminate? If so, in what state?

```
X := 5;
Y := 0;
while X >= 5 do
    if X = 5 then break else Y := 3 end
end
```

☐ Does not terminate

☒ Terminates with

```
X  =  5              Y  =  0
```

4.3 (2 points) Does the following program terminate? If so, in what state?

```
X := 0;
Y := 0;
while X < 10 do
    while Y < 10 do
        X = X + 1;
        break
    end
end
```

☐ Does not terminate

☒ Terminates with

```
X  =  10             Y  =  0
```

4.4 (10 points) Using the English description above as a guide, complete the formal definition of the `ceval` relation.

```
Definition if_then_else {A : Type} (b : bool) (l r : A) :=
  if b then l else r.

Inductive ceval : com -> state -> result -> state -> Prop :=
  | E_Skip : forall st,
      st =[ CSkip ]=> st / SContinue
  | E_Break : forall st,        (* FILL IN HERE... *)
```

Answer:

```
      st =[ CBreak ]=> st / SBreak

  | E_Asgn  : forall st a n x,
      aeval st a = n ->
      st =[ x := a ]=> (x !-> n ; st) / SContinue
  | E_SeqContinue : forall c1 c2 st st' st'' s,
      st =[ c1 ]=> st' / SContinue ->
      st' =[ c2 ]=> st'' / s ->
      st =[ c1 ; c2 ]=> st'' / s
  | E_SeqBreak : forall c1 c2 st st',   (* FILL IN HERE... *)
```

Answer:

```
      st =[ c1 ]=> st' / SBreak ->
      st =[ c1 ; c2 ]=> st' / SBreak

  | E_If : forall c1 c2 b st st' s,
      st =[ if_then_else (beval st b) c1 c2 ]=> st' / s ->
      st =[ if b then c1 else c2 end ]=> st' / s
  | E_WhileFalse : forall c b st,
      beval st b = false ->
      st =[ while b do c end ]=> st / SContinue
  | E_WhileContinue : forall c b st st' st'',     (* FILL IN HERE... *)
```

Answer:

```
      beval st b = true ->
      st  =[ c ]=> st' / SContinue ->
      st' =[ while b do c end ]=> st'' / SContinue ->
      st  =[ while b do c end ]=> st'' / SContinue

  | E_WhileBreak : forall c b st st',             (* FILL IN HERE... *)
```

Answer:

```
      beval st b = true ->
      st =[ c ]=> st' / SBreak ->
      st =[ while b do c end ]=> st' / SContinue
  where "st '=[' c ']=>' st' '/' s" := (ceval c st s st').
```

5 Big- and Small-step Semantics (25 points total)

Consider a tiny abstract machine that prints sequences of numbers. Its programs look like this:

```
Inductive prog : Type :=
  | Skip : prog                 (* Do nothing *)
  | Seq : prog -> prog -> prog  (* Sequence two programs *)
  | Print : nat -> prog.        (* Print a singular number *)
```

We can define its operational semantics as a *function* from programs to lists of numbers like this:

```
Fixpoint runf (p: prog) : list nat :=
  match p with
  | Skip => []
  | Seq p1 p2 => runf p1 ++ runf p2
  | Print n => [n]
  end.
```

Some examples:

- `runf Skip = []`
- `runf (Print 10) = [10]`
- `runf (Seq (Seq (Print 0) (Print 1)) (Seq (Skip) (Print 2))) = [0;1;2]`

5.1 (8 points) Another way to define the operational semantics of this language is as a *big-step inductive relation* between programs and lists of numbers. Of course, this alternative semantics should agree with the original one—that is, the following lemma should hold:

```
Lemma run_runf_eqv: forall p l, runf p = l <-> run p l.
```

Complete the definition of `run` below.

```
Inductive run : prog -> list nat -> Prop :=

| RSkip :
    run Skip []
| RSeq : forall p1 p2 o1 o2,
    run p1 o1 ->
    run p2 o2 ->
    run (Seq p1 p2) (o1 ++ o2)
| RPrint : forall n,
    run (Print n) [n].
```

5.2 (9 points) Yet another way to define the operational semantics of this language is as a *small-step relation* between a starting program together with a starting list of numbers (that have already been printed) and a "stepped" program together with a possibly-updated list of printed numbers.

- `forall l r, ~ step (Skip, l) r (* I.e., [Skip] never steps by itself. *)`
- `step (Print 2, [0;1]) (Skip, [0;1;2])`
- `step (Seq Skip (Print 1), [0]) (Print 1, [0])`
- ```
  step
    (Seq (Print 1) (Print 2), [0])
    (Seq (Skip)    (Print 2), [0;1])
  ```

Complete the following definition.

```
Inductive step : (prog * list nat) -> (prog * list nat) -> Prop :=
| SSeqSkip : forall l p2,
  step (Seq Skip p2, l) (p2, l)
| SSeqCong : forall p1 p1' p2 l l',
  step (p1, l) (p1', l') ->
  step (Seq p1 p2, l) (Seq p1' p2, l')
| SPrint : forall n l,
  step (Print n, l) (Skip, l ++ [n]).
```

5.3 (8 points) Which of the following statements are provable?

(a) `forall p, exists ls, run p ls.`

☒ True ☐ False

(b) `forall ls, exists p, run p ls.`

☒ True ☐ False

(c)
```
forall p ls ls',
  run p ls ->
  run p ls' ->
  ls = ls'.
```

☒ True ☐ False

(d)
```
forall p ls ls',
  multi step (p, ls) (Skip, ls') ->
  run p ls'.
```

☐ True ☒ False

(e)
```
forall p l0 p' l1,
  step (p, l0) (p', l1) ->
  exists ext, l1 = l0 ++ ext.
```

☒ True ☐ False
```

For Reference

Big-Step Semantics of Imp

```
                  -----------------                                  (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