

CIS 500: Software Foundations

Midterm II

November 18, 2021

Solutions

1 Program Equivalence (12 points)

The Equiv chapter introduced the concept of program equivalence:

- Two 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 example,

```
while X < 100 do X := X - 1; X := X - 1 end
```

and

```
while X < 100 do X := X - 2 end
```

are equivalent.

In this problem, we will work with some related notions:

- Commands c_1 and c_2 are *equivalent modulo termination* if, for every starting state st , if c_1 terminates in state $st'1$ and c_2 terminates in state $st'2$, then $st'1 = st'2$.

“Equivalent modulo termination” means that both commands yield the same final state whenever they both terminate (but they do not necessarily both terminate or both diverge on a given starting state).

For example,

```
while true do skip end
```

and

```
X := 5
```

are equivalent modulo termination.

- Commands c_1 and c_2 are *sometimes different* if, for some starting state st , command c_1 terminates in state $st'1$ and c_2 terminates in state $st'2$, with $st'1 \neq st'2$.

For example,

```
if X = 0 then Y := 5 else skip end
```

and

```
if X = 0 then Y := 42 else skip end
```

are sometimes different.

- Commands c_1 and c_2 are *always different* if, for every starting state st such that c_1 terminates in state $st'1$ and c_2 terminates in state $st'2$, we have $st'1 \neq st'2$.

For example,

```
X := X + 1
```

and

```
X := X + 2
```

are always different.

Also:

- A command C is *total* if it terminates on all input states – i.e., if, for every st , running c starting in state st yields some final state st' .

For example,

```
while X > 10 do X := X - 1 end
```

is total, but

```
while X > 10 do X := X + 1 end
```

is not.

Let's begin with a few warmup questions to clarify the relations among all these concepts...

- 1.1 Two commands that are *equivalent* are also *equivalent modulo termination*.
 True False
- 1.2 Two commands that are *sometimes different* are also *always different*.
 True False
- 1.3 Two commands that are *equivalent modulo termination* and are both *total* are *equivalent*.
 True False
- 1.4 It is possible for two commands to be simultaneously *equivalent* and *sometimes different*.
 True False
- 1.5 If two commands are *not equivalent*, then they are *sometimes different*.
 True False
- 1.6 If two *total* commands are *not equivalent*, then they are *sometimes different*.
 True False

2 Program Equivalence, continued (10 points)

For each of the following pairs of programs, mark the term that best describes how they are related. (If the programs are both Equivalent and Equivalent Modulo Termination, mark just Equivalent.)

2.1

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

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

- Equiv Eqv mod term Sometimes diff Always diff

2.2

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

```
X := 42;  
while X > 0 do X := X-1 end
```

- Equiv Eqv mod term Sometimes diff Always diff

2.3

```
X := 1;  
while X > 10 do X := X+1 end
```

```
X := 42;  
while X > 10 do X := X+1 end
```

- Equiv Eqv mod term Sometimes diff Always diff

2.4

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

```
while X > 42 do X := X+1 end
```

- Equiv Eqv mod term Sometimes diff Always diff

2.5

```
while true do X := X + 1 end
```

```
while true do X := X - 1 end
```

- Equiv Eqv mod term Sometimes diff Always diff

3 [Standard Track Only] Loop Invariants (18 points)

Recall that a *loop invariant* for a while loop

```
while b do c end
```

is an assertion P such that $\{P \wedge b\} c \{P\}$ is a valid Hoare triple. (To be useful in a larger proof, we would also want that P is implied by whatever was true before the loop and that P implies whatever we need to be true after the loop. For present purposes, we are ignoring these conditions and focusing on just the loop body.)

For example, the assertion $X = Y/2$ is an invariant of the loop

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

because the triple

```
{X = Y/2 /\ Y > 1 }
X := X - 1;
Y := Y - 2
{X = Y/2 }
```

is valid, but $X = Y$ is not an invariant of this loop, because the triple

```
{X = Y /\ Y > 1 }
X := X - 1;
Y := Y - 2
{X = Y }
```

is not valid.

For each loop shown below, check the “Yes” box next to each assertion that is a valid loop invariant. Check the “No” box next to those that are not.

3.1 while $X < 100$ do
 $X := X + 1$;
 $Y := Y - 1$
end

- | | | |
|---|--|--------------|
| <input checked="" type="checkbox"/> Yes | <input type="checkbox"/> No | $X > 10$ |
| <input type="checkbox"/> Yes | <input checked="" type="checkbox"/> No | $Y > 10$ |
| <input checked="" type="checkbox"/> Yes | <input type="checkbox"/> No | $X \leq 100$ |

3.2 while X>100 do
 X := X+1;
 Y := Y-1
end

- | | | |
|---|--|--------|
| <input checked="" type="checkbox"/> Yes | <input type="checkbox"/> No | X > 10 |
| <input type="checkbox"/> Yes | <input checked="" type="checkbox"/> No | Y > 10 |
| <input checked="" type="checkbox"/> Yes | <input type="checkbox"/> No | True |
| <input checked="" type="checkbox"/> Yes | <input type="checkbox"/> No | False |

3.3 while X<100 do
 if X<Y then
 X := 0
 else
 X := X + 1
 fi
end

- | | | |
|---|-----------------------------|----------|
| <input checked="" type="checkbox"/> Yes | <input type="checkbox"/> No | X > 1000 |
| <input checked="" type="checkbox"/> Yes | <input type="checkbox"/> No | Y > 10 |

4 Strongest Postconditions (15 points)

Suppose we are given a command c and some precondition P . In general, there may be many postconditions Q 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 Q that is *stronger* than all the others—i.e., such that $Q \rightarrow Q'$ whenever $\{P\} c \{Q'\}$ is valid.

For example, these are all valid triples,

```
{ X=1 } X := X + 1 { X=2 }
{ X=1 } X := X + 1 { X>1 }
{ X=1 } X := X + 1 { True }
```

but $X=2$ is the strongest postcondition for this command and precondition.

Complete the following triples with their strongest postconditions.

4.1 $\{ Y = 5 \} X := Z \{ Q \}$

Answer:

$Q = X = Z \wedge Y = 5$

4.2 $\{ \text{True} \} X := X + 1 \{ Q \}$

Answer:

$Q = X > 0$

4.3 $\{ X = m \wedge Y = n \} Z := X; X := Y; Y := Z \{ Q \}$

Answer:

$Q = X = n \wedge Y = m \wedge Z = m$

4.4 $\{ Y > 100 \} \text{ while } 0 \leq Y \text{ do } Y := Y + 1 \text{ end } \{ Q \}$

Answer:

$Q = \text{False}$

4.5 $\{ \text{True} \} \text{ if } Y = 0 \text{ then } X := 1 \text{ else } X := Y \{ Q \}$

Answer:

$Q = (Y=0 \wedge X=1) \vee (Y>0 \wedge X=Y)$

5 **Big-step and Small-step operational semantics** (17 points)

In this problem we will work with a simple register machine with an infinite set of registers, each of which can either be *uninitialized* or hold a number. It has three instructions: assignment of a `nat` to a register, copying a value from one register to another, and adding the number in one register into another register.

The evaluation function `evalf` below gives a precise specification of how the machine behaves.

```
Definition reg := nat.
```

```
Inductive rinstr : Type :=
| SAsgn (n: reg)(v: nat)
| SCopy (from to: reg)
| SAdd (from to: reg).
```

```
Definition registers := partial_map nat.
```

```
Fixpoint evalf (prog : list rinstr) (st: registers) : option registers :=
  match prog with
  | [] => Some st
  | (SAsgn n v) :: rest => evalf rest (n |-> v; st)
  | (SCopy f t) :: rest => match st f with
    | Some fv => evalf rest (t |-> fv; st)
    | _ => None
    end
  | (SAdd f t) :: rest => match st f, st t with
    | Some v1, Some v2 => evalf rest (t |-> v1 + v2; st)
    | _, _ => None
    end
  end.
```

For example, the program `[SCopy 0 1; SAdd 1 0]` first copies the contents of register 0 to register 1 and then adds register 1 back into register 0. If we run this program from a starting state where register 0 has value 42, we get this:

```
evalf [SCopy 0 1; SAdd 1 0]
      (0 |-> 42)
= Some (0 |-> 84; 1 |-> 42)
```

Notice that the `evalf` function returns an `option`—i.e., evaluation can *fail*. In particular, the `SCopy` operation fails (returning `None`) if its source register is uninitialized, and the `SAdd` operation fails if either register is uninitialized. For example, if we try to copy register 5 into register 6 but only register 0 is initialized, the program will fail:

```
evalf [SCopy 5 6]
      (0 |-> 42)
= None
```

Your job will be to fill in the details of two inductively defined relations—a big-step evaluation relation and a small-step reduction relation—so that they capture the same behavior.

- 5.1 Complete this inductively defined relation for the big-step semantics. For example, the following should be provable using the relation you define:

```
bstep [SCopy 0 1; SAdd 1 0] (0 |-> 42) (0 |-> 84; 1 |-> 42).
```

On the other hand,

```
bstep [SCopy 5 6] (0 |-> 42) st'.
```

should *not* be provable for any st' .

```
Inductive bstep: list rinstr -> registers -> registers -> Prop :=
| BAsgn: forall ts st st' n v,
  bstep ts st st' ->
  bstep (SAsgn n v :: ts) st (n |-> v; st')
| BCopy: forall f t ts st st' fv tv,
  bstep ts st st' ->
  Some fv = st' f ->
  Some tv = st' t ->
  bstep (SCopy f t :: ts) st (t |-> fv; st')
| BAdd: forall ts st st' av bv a b,
  bstep ts st st' ->
  Some av = st' a ->
  Some bv = st' b ->
  bstep (SAdd a b :: ts) st (b |-> av + bv; st').
```

- 5.2 Complete this inductively defined relation for the small-step semantics. For example, the following should be provable using the relation you define:

```
sstep
  [SCopy 0 1; SAdd 1 0] (0 |-> 42)
  [SAdd 1 0] (0 |-> 42; 1 |-> 42)
```

On the other hand,

```
sstep [SCopy 5 6] (0 |-> 42) st'.
```

should *not* be provable for any st' .

```
Inductive sstep: list rinstr -> registers -> list rinstr -> registers -> Prop :=
| SSAsgn: forall ts st n v,
  sstep (SAsgn n v :: ts) st ts (n |-> v; st)
| SSCopy: forall fv tv f t ts st,
  Some fv = st f ->
  Some tv = st t ->
  sstep (SCopy f t :: ts) st ts (t |-> fv; st)
| SSPlus: forall a1v a2v a1 a2 ts st,
  Some a1v = st a1 ->
  Some a2v = st a2 ->
  sstep (SAdd a1 a2 :: ts) st ts (a2 |-> a1v + a2v; st).
```

6 Types (18 points)

Here is a simple set of terms, just like the ones we saw in the SmallStep chapter, except that instead of constants and addition we now have constants (C) and *subtraction* (M).

```
Inductive tm :=
  | C (n : nat)
  | M (t1 t2 : tm).
```

The `step` relation for this language can be defined as in SmallStep, simply replacing addition with subtraction:

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

```
Inductive step : tm -> tm -> Prop :=
  | ST_MinusConstConst : forall v1 v2,
    M (C v1) (C v2) --> C (v1 - v2)
  | ST_Minus1 : forall t1 t1' t2,
    t1 --> t1' ->
    M t1 t2 --> M t1' t2
  | ST_Minus2 : forall v1 t2 t2',
    t2 --> t2' ->
    M (C v1) t2 --> M (C v1) t2'
```

```
where " t '-->' t' " := (step t t').
```

Now suppose we equip this language with the following slightly unusual type system. The set of types has two elements

```
Inductive ty :=
  | TZ
  | TU.
```

(pronounced “zero” and “unknown”) and the typing relation is defined as follows:

```
Reserved Notation "'|-' t '\in' T" (at level 40).
```

```
Inductive has_type : tm -> ty -> Prop :=
  | TCOZ :
    |- C 0 \in TZ
  | TCOU : forall n,
    |- C n \in TU
  | TMZ : forall t1 t2 T,
    |- t1 \in T ->
    |- t2 \in TZ ->
    |- M t1 t2 \in T
  | TMU : forall t1 t2,
    |- t1 \in TU ->
    |- t2 \in TU ->
    |- M t1 t2 \in TU
```

```
where "'|-' t '\in' T" := (has_type t T).
```

6.1 Is the **step** relation in this language deterministic (i.e., does every term step to at most one other term)?

Yes No

6.2 Is the **step** relation in this language total (i.e., for every t is there some t' such that $t \rightarrow t'$)?

Yes No

6.3 Is the **step** relation for well-typed terms total (i.e., for every t with $\vdash t \text{ in } T$ for some T is there some t' such that $t \rightarrow t'$)?

Yes No

6.4 Is there a term t in this language that has no types (i.e., such that there is no type T with $\vdash t \text{ in } T$)?

Yes No

If you answered Yes, give an example. If you answered No, briefly explain why not.

Every term has type TU.

6.5 Is there a term t in this language that has *multiple* types (i.e., such that $\vdash t \text{ in } T_1$ and $\vdash t \text{ in } T_2$ where T_1 and T_2 are different)?

Yes No

If you answered Yes, give an example. If you answered No, briefly explain why not.

C 0 has both type TZ and type TU.

6.6 Can a term lose types as it steps? I.e., are there terms t_1 and t_1' and a type T such that $t_1 \rightarrow t_1'$, where $\vdash t_1 \text{ in } T$ but not $\vdash t_1' \text{ in } T$?

Yes No

If you answered Yes, give an example. If you answered No, briefly explain why not.

Every term has type TU, so the only way a term t_1 could lose a type would be if it had type TZ but stepped to a term t_1' that did not have type TZ. The only terms of type TZ are C 0 and terms of the form M t1 t2 where t1 has type TZ — i.e., they look like M (M (M ... (M (C 0) ...))). Stepping a term of this shape always yields another of the same shape. .

6.7 Can a term *gain* types as it steps? I.e., are there terms t_1 and t_1' and a type T such that $t_1 \rightarrow t_1'$, where $\vdash t_1' \in T$ but not $\vdash t_1 \in T$?

Yes No

If you answered Yes, give an example. If you answered No, briefly explain why not.

$M(C\ 2)\ (C\ 2)$ does not have type TZ , but it steps to $C\ 0$, which does.

7 [Advanced Track Only] Imp + Havoc (Informal proof) (18 points)

Recall the HImp (Imp + Havoc) language from the Equiv chapter. The definition of the big-step evaluation relation for this language can be found in the appendix.

A command c_1 is said to *refine* c_2 if the possible ending states of c_1 are a subset of the possible ending states of c_2 for every starting state. Formally:

```
Definition refines (c1 c2 : com) : Prop :=
  forall (st st' : state),
    (st =[ c1 ]=> st') -> (st =[ c2 ]=> st').
```

(It is exactly the same as the usual command equivalence, `cequiv`, except that “iff” is replaced here by “implies”.)

7.1 Theorem: Suppose c_1 and c_2 are HImp programs such that c_1 refines c_2 . Then `while b do c1 end` refines `while b do c2 end`.

Proof:

Suppose that $st = [\text{while } b \text{ do } c_1 \text{ end}] \Rightarrow st'$. We must show that $st = [\text{while } b \text{ do } c_2 \text{ end}] \Rightarrow st'$.

Proceed by induction on the given derivation of $st = [\text{while } b \text{ do } c_1 \text{ end}] \Rightarrow st'$.

If the last step in the derivation is a use of the `E_WhileFalse` rule, then $st = st'$ and $st \ b = \text{false}$. The result follows immediately by `E_WhileFalse`.

Suppose the last in the derivation is a use of the `E_WhileTrue` rule, with

```
st b = true
st =[c1]=> st''
st'' =[while b do c1 end]=> st'
st'' =[while b do c2 end]=> st'   (the IH).
```

Since c_1 refines c_2 , we have $st = [c_1] \Rightarrow st''$. Putting this together with the IH using rule `E_WhileTrue` gives us $st = [\text{while } b \text{ do } c_2 \text{ end}] \Rightarrow st'$, as required.

7.2 Consider the following two programs:

```
c42 =
  havoc X;
  X := X + 42
```

```
c5 =
  havoc X;
  X := X + 5
```

Show that c42 refines c1.

Proof:

By inspecting the evaluation rules, we see that a derivation of $st = [c42] \Rightarrow st'$ must consist of a use of E_Seq

$$\begin{aligned} st &= [\text{havoc } X] \Rightarrow st1 \\ st1 &= [X := X + 42] \Rightarrow st' \end{aligned}$$

joining a use of E_Havoc and a use of E_Asgn:

$$\begin{aligned} st1 &= X \mid \rightarrow n; st && \text{for some } n \\ st' &= X \mid \rightarrow (st1 \ X) + 42; st \\ &= X \mid \rightarrow n + 42; st \end{aligned}$$

We can build a derivation for $st = [c1] \Rightarrow st'$ as follows, using E_Havoc, E_Asgn, and E_Seq:

$$\begin{array}{ccc} st = [\text{havoc } X] \Rightarrow (X \mid \rightarrow n + 41; st) & (X \mid \rightarrow n + 41; st) = [X := X + 1] \Rightarrow st' & \\ \hline & st = [c1] \Rightarrow st' & \end{array}$$

For Reference

Hoare logic

Definition hoare_triple
(P : Assertion) (c : com) (Q : Assertion) : Prop :=
forall st st',
 st =[c]=> st' ->
 P st ->
 Q st'.

Notation " $\{\{ P \}\}$ c $\{\{ Q \}\}$ " :=
(hoare_triple P c Q) (at level 90, c custom com at level 99)
: hoare_spec_scope.

Axiom hoare_skip : forall P,
 $\{\{P\}\}$ skip $\{\{P\}\}$.

Axiom hoare_seq : forall P Q R c1 c2,
 $\{\{Q\}\}$ c2 $\{\{R\}\}$ ->
 $\{\{P\}\}$ c1 $\{\{Q\}\}$ ->
 $\{\{P\}\}$ c1; c2 $\{\{R\}\}$.

Axiom hoare_asgn : forall Q X a,
 $\{\{Q [X \mapsto a]\}\}$ X := a $\{\{Q\}\}$.

Axiom hoare_seq : forall P Q R c1 c2,
 $\{\{Q\}\}$ c2 $\{\{R\}\}$ ->
 $\{\{P\}\}$ c1 $\{\{Q\}\}$ ->
 $\{\{P\}\}$ c1;c2 $\{\{R\}\}$.

Axiom hoare_skip : forall P,
 $\{\{P\}\}$ skip $\{\{P\}\}$.

Axiom hoare_consequence : forall (P P' Q Q' : Assertion) c,
 $\{\{P'\}\}$ c $\{\{Q'\}\}$ ->
P ->> P' ->
Q' ->> Q ->
 $\{\{P\}\}$ c $\{\{Q\}\}$.

Axiom hoare_if : forall P Q (b:bexp) c1 c2,
 $\{\{ P \wedge b \}\}$ c1 $\{\{Q\}\}$ ->
 $\{\{ P \wedge \sim b \}\}$ c2 $\{\{Q\}\}$ ->
 $\{\{P\}\}$ if b then c1 else c2 end $\{\{Q\}\}$.

Axiom hoare_while : forall P (b:bexp) c,
 $\{\{P \wedge b\}\}$ c $\{\{P\}\}$ ->
 $\{\{P\}\}$ while b do c end $\{\{ P \wedge \sim b \}\}$.

Imp + Havoc

```
Inductive ceval : com -> state -> state -> Prop :=
| E_Skip : forall st,
  st =[ skip ]=> st
| E_Asgn  : forall st a n x,
  aeval st a = n ->
  st =[ x := a ]=> (x !-> n ; st)
| E_Seq   : forall c1 c2 st st' st'',
  st  =[ c1 ]=> st'  ->
  st' =[ c2 ]=> st'' ->
  st  =[ c1 ; c2 ]=> st''
| E_IfTrue : forall st st' b c1 c2,
  beval st b = true ->
  st  =[ c1 ]=> st'  ->
  st  =[ if b then c1 else c2 end ]=> st'
| E_IfFalse : forall st st' b c1 c2,
  beval st b = false ->
  st  =[ c2 ]=> st'  ->
  st  =[ if b then c1 else c2 end ]=> st'
| E_WhileFalse : forall b st c,
  beval st b = false ->
  st  =[ while b do c end ]=> st
| E_WhileTrue : forall st st' st'' b c,
  beval st b = true ->
  st  =[ c ]=> st'  ->
  st' =[ while b do c end ]=> st'' ->
  st  =[ while b do c end ]=> st''
| E_Havoc : forall (st : state) (x : string) (n : nat),
  st =[ havoc x ]=> (x !-> n ; st)
```