CIS 500 — Software Foundations

Midterm II

Answer key

March 31, 2010

1. (4 points) The syntax and big-step semantics of IMP programs (over numbers and lists) are provided for your reference in the Appendix. Throughout the exam, we use informal notations in programs to make them easier to read; for example, we write `3 + 3` instead of `APlus (ANum 3) (ANum 3)` and `h :: t` instead of `ACons h t`.

   Recall the definition of equivalence for IMP programs:

```
Definition cequiv (c1 c2 : com) : Prop :=
  forall (st st':state), (c1 / st ==> st') ↔ (c2 / st ==> st').
```

   Which of the following pairs of programs are equivalent? Write "yes" or "no" for each one. (Where it appears, `a` is an arbitrary `aexp` — i.e., you should write "yes" only if the two programs are equivalent for every `a`.)

   (a)
```
Y ::= a :: X
```
   and
```
Y ::= a :: (head X) :: (tail X)
```
   Answer: No; consider when X starts out as nil.

   (b)
```
WHILE IsCons Z DO
  Z ::= tail Z
END
```
   and
```
Z ::= nil
```
   Answer: No. For example, if Z has the value 0, the first program leaves it as 0 (since 0 is interpreted as Nil) but the second program sets it to nil.

   (c)
```
WHILE X <> 0 DO
  X ::= X + 1
END
```
   and
```
WHILE X <> 0 DO
  Y ::= Y + 1
END
```
   Answer: Yes

(d)
```
Y ::= 0;
WHILE 0 <= X DO
  Y ::= Y + 3;
  X ::= X - 1
END
```
and
```
Y ::= 3 * X;
X ::= 0
```
Answer: No. The first program does not terminate.

2. (10 points) Indicate whether or not each of the following Hoare triples is valid by writing either "valid" or "invalid." Also, for those that are invalid, give a counter-example. Where it appears, `a` is an arbitrary `aexp`—i.e., you should write "valid" only if the triple is valid for every `a`.

(a)
```
{ True } X ::= a { X = a }
```
Answer: Invalid. For example, is not satisfied when `a = X + 1`.

(b)
```
{ X <= Y }
  IF Y == 0 THEN Y ::= 3 ELSE X ::= 0 FI
{ X = 0 }
```
Answer: Valid

(c)
```
{ X = h :: t }
  Y ::= 0;
  WHILE IsCons X DO
    IF Y == 0 THEN
      Y ::= head X
    ELSE
      SKIP
    FI;
    X ::= tail X
  END
{ Y = h }
```
Answer: Invalid. For example, if `X = 0 :: 3 :: []`, *then* `Y` *will end up as* `3` *rather than* `0`.

(d)
```
{ 1 <= Y ∧ Y = a }
  Z ::= 0;
  WHILE Y <> 0 DO
    Z ::= Z + Y;
    X ::= X - 1
  END
{ Z = a * a }
```
Answer: Valid; the loop does not terminate.

(e)
```
{ True }
  WHILE Z <> 2 DO
    Z ::= Z + 2
  END
{ False }
```
Answer: Invalid; the loop does terminate if Z starts out as 0.

3. (16 points) Recall the simple expression language with plus and numeric constants, which we introduced in Smallstep.v. (It is reproduced in the Appendix with both big-step and small-step semantics.)

   The following theorem captures the intuition that "big-step reduction implies small-step."

   Theorem: For all `t` and `v`, if `t ==> v` then `t -->* v`.

   In the space below, write a careful proof of this theorem in English. You may use the following lemmas without proving them:

```
Lemma stepmany_congr_1 : forall t1 t1' t2,
    t1 -->* t1' →
    tm_plus t1 t2 -->* tm_plus t1' t2.

Lemma stepmany_congr_2 : forall t1 t2 t2',
    value t1 →
    t2 -->* t2' →
    tm_plus t1 t2 -->* tm_plus t1 t2'.

Lemma rsc_trans : forall (X:Type) (R: relation X) (x y z : X),
    refl_step_closure R x y  →
    refl_step_closure R y z →
    refl_step_closure R x z.
```

   Answer: Proof: By induction on a derivation of `t ==> v`.

   - Suppose the final rule used to show `t ==> v` is `E_Const`. Then `t = tm_const n = v`. We must show `stepmany (tm_const n) (tm_const n)`. This holds by `rsc_refl`.
   - Suppose the final rule used to show `t ==> v` is `E_Plus`. Then `t = tm_plus t1 t2`, and we know that `t1 ==> tm_const n1` and `t2 ==> tm_const n2` for some `n1` and `n2`. The IH tells us that `t1 -->* tm_const n1` and `t2 -->* tm_const n2`. We must show that `tm_plus t1 t2 -->* tm_const (plus n1 n2)`.

     First, notice that

```
tm_plus t1 t2 -->* tm_plus (tm_const n1) t2
```

     by `stepmany_congr_1` and the `stepmany` derivation for `t1`. Observing that `tm_const n1` is a value, we also notice

```
tm_plus (tm_const n1) t2 -->*
tm_plus (tm_const n1) (tm_const n2)
```

     by `stepmany_congr_2` and the `stepmany` derivation for `t2`. It's also easy to see by `ST_PlusConstConst` that

```
tm_plus (tm_const n1) (tm_const n2) -->
tm_const (plus n1 n2)
```

     and so, by `rsc_step` and `rsc_refl`, that the same is true for `-->*`. We can now use transitivity of `-->*` to stitch these derivations, proving

```
tm_plus t1 t2 -->* tm_const (plus n1 n2).
```

   Grading scheme: 0-6 points for missing or very garbled proofs, or for proofs of the wrong theorem. 7-11 points for proofs that included most of the important ideas but weren't put together completely correctly. 12-16 points for correct proofs, perhaps with small problems.

4. (8 points) Conversely, we also saw in `Smallstep.v` that "small-step implies big-step." State the theorem (in formal Coq notation) that expresses this precisely. (Do not give a proof — just the statement of the theorem.)

   Answer:

```
Theorem stepmany__eval : forall t v,
  normal_form_of t v → t ==> v.
```

   Grading scheme: -1 each for misapplication of -> and ==>. -1 for requiring a value instead of a normal form. -4 for failing to constrain the small-step reduct at all. -1 for Coq syntax errors.*

5. (10 points) Recall the definition of stack machine programs and their small-step semantics (omitting the `SMinus` and `SMult` instructions):

```
Inductive sinstr : Type :=
| SPush : nat → sinstr
| SLoad : id → sinstr
| SPlus : sinstr.

Definition stack := list nat.
Definition prog  := list sinstr.

Inductive stack_step : state → prog * stack → prog * stack → Prop :=
  | SS_Push : forall st stk n p',
      stack_step st (SPush n :: p', stk)      (p', n :: stk)
  | SS_Load : forall st stk i p',
      stack_step st (SLoad i :: p', stk)      (p', st i :: stk)
  | SS_Plus : forall st stk n m p',
      stack_step st (SPlus :: p', n::m::stk)  (p', (m+n)::stk).
```

   Suppose we want stack machine programs to also support a *store* instruction:

```
Inductive sinstr : Type :=
  ...      (* same instructions as before... *)
| SStore : id → sinstr.
```

   Write down a new version of `stack_step` that handles the new instruction. Executing `SStore X` should have the effect of popping a value from the top of the stack and assigning it to `X`.

   Answer:

```
Inductive stack_step : prog * (state * stack) → prog * (state * stack) → Prop :=
  | SS_Push : forall st stk n p',
    stack_step (SPush n :: p', (st, stk))      (p', (st, n::stk))
  | SS_Load : forall st stk i p',
    stack_step (SLoad i :: p', (st, stk))      (p', (st, st i :: stk))
  | SS_Plus : forall st stk n m p',
    stack_step (SPlus :: p', (st, n::m::stk))  (p', (st, (m+n)::stk))
  | SS_Store : forall st stk n p',
    stack_step (SStore x :: p', (st, n::stk))  (p', (update st x n, stk)).
```

   Grading scheme: 3 points for mentioning (update st x n) or something of the sort. 4 points for adding an output state to the type of the step relation. 3 points for implementing the rules correctly.

6. (9 points) Recall the `cstep` relation, which defines the small-step variant of the operational semantics of IMP (the full definition is given in the Appendix).

   Consider extending the syntax of IMP programs with a new primitive command called `ANYTHING`:

```
Inductive com : Type :=
...
| CAnything : com.

Notation "'ANYTHING'" := CAnything.
```

   The behavior of `ANYTHING` is completely nondeterministic: after it executes, the memory can be left in any state whatsoever. However, `ANYTHING` is guaranteed to always terminate.

   (a) What needs to be added to the definition of `cstep` to give `ANYTHING` this behavior?
   Answer:

```
| CS_Anything : forall st st', ANYTHING / st --> SKIP / st'.
```

   (b) Write down a Hoare rule for reasoning about `ANYTHING`, and explain why it is the right rule.
   Answer: The right rule is `{ True } ANYTHING { True }`. *We can't say anything about the final state, so the only thing we can put as the postcondition is* `True`. *Any precondition will make the triple valid, so we may as well put the weakest possible precondition, namely* `True`.

   Grading scheme: Distribution of points: part (a) 4 points, part (b) 5 points. Most common mistake on part (a) was forgetting to write SKIP as the result of the step: -2 points. Part (b): 1 point for writing the correct rule; 4 points for correct explanation, partial credit was given.

7. (15 points) Recall the definition of the `length` function from Basics.v:

```
Fixpoint length (xs:natlist) : nat :=
  match xs with
  | nil => O
  | h :: t => S (length t)
  end.
```

Here is a small Imp program that implements the same calculation:

```
                          {X = xs}
Y ::= 0;
WHILE X <> [] DO
  Y := Y + 1;
  X ::= tail(X)
END
                          {Y = length xs}
```

In the space below, write a fully decorated version of this program demonstrating that it validates the stated pre- and post-conditions.

Answer:

```
                          {X = l} =>
                          {0 + length X = length l}
Y ::= 0;
                          {Y + length X = length l}
WHILE X <> [] DO
                          {Y + length X = length l ∧ X <> []} =>
                          {Y + 1 + length (tail X) = length l}
  Y := Y + 1;
                          {Y + length (tail X) = length l}
  X ::= tail(X)
                          {Y + length X = length l}
END
                          {Y + length X = length l ∧ ~(X <> [])} =>
                          {Y = length l}
```

Grading scheme: Each line of the program was worth a maximum of 3 points for applying the appropriate Hoare rule correctly. Incorrect uses of the assignment rule lost 3 points the first time and 2 points each subsequent time. Partial credit was given in individual cases.

8. (8 points) Here is a Coq function that returns `true` iff its argument is a list containing only zeros.

```
Fixpoint allzero (xs:natlist) : bool :=
  match xs with
  | nil => true
  | h :: t => andb (beq_nat h 0) (allzero t)
  end.
```

Here is an Imp program that performs the same test: when it terminates, the variable `Y` will be set to `0` iff the variable `X` at the beginning contains only zeros. If we want to prove that the program behaves as specified — *i.e.*, that the program together with the indicated pre- and post-conditions forms a valid Hoare triple — what invariant would we need to use for the loop; (*i.e.*, what assertion should we write in place of `???`)?

Answer:

```
                          {Y = 0 ↔ allzero (rev Z) ∧ (rev Z)++X = xs}
```

Grading scheme: -1 for allzero Z instead of allzero (rev Z) (waived if below 6 already). -5 (as appropriate) for over/under specification of how much of X/Z/xs are nonzero iff Y = 0. -3 for failing to constrain X/Z/xs. (There was a fair bit of variation in answers to this one. A lot of people forgot one or another conjunct, resulting in 5/8. Others tried to constrain Y and X directly, or even Y and xs! We deducted -5 or more in this case.)