CIS 500 — Software Foundations

Final Exam

May 9, 2011

Answer key

Hoare Logic

1. (7 points) What does it mean to say that the Hoare triple `{{P}} c {{Q}}` is *valid*?

Answer: `{{P}} c {{Q}}` means that, for any states `st` and `st'`, if `st` satisfies `P` and `c / st` $\Downarrow$ `st'`, then `st'` satisfies `Q`.

2. (18 points) Recall the Hoare rule for reasoning about sequences of commands:

$$\frac{\{\{\texttt{P}\}\}\ \texttt{c1}\ \{\{\texttt{Q}\}\} \qquad \{\{\texttt{Q}\}\}\ \texttt{c2}\ \{\{\texttt{R}\}\}}{\{\{\texttt{P}\}\}\ \texttt{c1;c2}\ \{\{\texttt{R}\}\}}\ \textsc{Hoare\_Seq}$$

Formally, this rule corresponds to a theorem:

```
Theorem hoare_seq : forall P Q R c1 c2,
     {{P}} c1 {{Q}} ->
     {{Q}} c2 {{R}} ->
     {{P}} c1;c2 {{R}}.
```

Give a careful informal proof (in English) of this theorem.

Answer: To show that the Hoare triple `{{P}} c1;c2 {{R}}` is valid, we must show that, for any states `st` and `st''`, if `st` satisfies `P` and `c1;c2 / st` $\Downarrow$ `st''`, then `st''` satisfies `R`. So suppose `st` satisfies `P` and `c1;c2 / st` $\Downarrow$ `st'`.

By the definition of the evaluation relation, there must be some state `st'` such that `c1 / st` $\Downarrow$ `st'` and `c2 / st'` $\Downarrow$ `st''`. Since `{{P}} c1 {{Q}}` is a valid triple, we know that `st'` satisfies `Q`. But then, since `{{Q}} c2 {{R}}` is a valid triple, we know that `st''` satisfies `R`, as required.

3. (12 points) In the `Imp` program below, we have provided a precondition and postcondition. In the blank before the loop, fill in an invariant that would allow us to annotate the rest of the program.

```
                          { True }
X := n
Y := X
Z := 0
                          { ____________________________________________ }
WHILE Y <> 0 DO
    Z := Z + X;
    Y := Y - 1
END
                          { Z = n*n }
```

Answer: `Z + Y*n = n*n` $\wedge$ `X = n`

Grading scheme: Common mistakes included

- *Using subtraction to state the invariant without including an assertion about the relative sizes of the things being subtracted (-2 points)*
- *Omitting* `X = n` *from the invariant (-3 points)*

Otherwise, 4 points each for:

- *invariant holds before the loop*
- *invariant is preserved by the loop*
- *invariant and negation of loop test implies final assertion*

STLC

4. (16 points) Recall the definition of the *substitution* operation in the simply typed lambda-calculus (with no extensions, and omitting base types such as booleans for brevity):

```
Fixpoint subst (s:tm) (x:id) (t:tm) : tm :=
  match t with
  | tm_app t1 t2 => tm_app (subst s x t1) (subst s x t2)
  | tm_var x' => if beq_id x x' then s else t
  | tm_abs x' T t1 => tm_abs x' T (if beq_id x x' then t1 else (subst s x t1))
  end.
```

This definition uses Coq's `Fixpoint` facility to define substitution as a *function*. Suppose, instead, we wanted to define substitution as an inductive *relation* `substi`. We've begun the definition by providing the `Inductive` header and one of the constructors; your job is to fill in the rest of the constructors. (Your answer should be such that `subst s x t = t' <-> substi s x t t'`, for all `s`, `x`, `t`, and `t'`, but you do not need to prove it).
Answer:

```
Inductive substi (s:tm) (x:id) : tm -> tm -> Prop :=
  | s_app : forall t1 t2 t1' t2',
      substi s x t1 t1' ->
      substi s x t2 t2' ->
      substi s x (tm_app t1 t2) (tm_app t1' t2')
  | s_var1 :
      substi s x (tm_var x) s
  | s_var2 : forall x',
      beq_id x x' = false ->
      substi s x (tm_var x') (tm_var x')
  | s_abs1 : forall T t1,
      substi s x (tm_abs x T t1) (tm_abs x T t1)
  | s_abs2 : forall x' T t1 t1',
      beq_id x x' = false ->
      substi s x t1 t1' ->
      substi s x (tm_abs x' T t1) (tm_abs x' T t1').
```

References

5. (12 points) The next few problems concern the STLC extended with natural numbers and references (reproduced on page 13, with the same informal notations as we're using here).

(a) In this system, is there a type `T` that makes

```
x:T; [] |- (\x:Nat. 2 * x) (x x) : Nat
```

provable? If so, what is it?

Answer: No.

(b) Is there a type `T` that makes

```
empty; [] |- (\x:Ref Nat. ((\_:Unit. !x), (\y:Nat. x := y))) (ref 0) : T
```

provable? If so, what is it?

Answer: `(Unit->Nat)*(Nat->Unit)`

(c) Is there a type `T` that makes

```
x:T; [] |- !(!(!x)) : Nat
```

provable? If so, what is it?

Answer: `Ref (Ref (Ref Nat))`

(d) Is there a type `T` that makes

```
x:T; [] |- (\y:Nat*Nat. pred (y.fst)) (x.snd x.fst) : Nat
```

provable? If so, what is it?

Answer: `S * (S -> Nat*Nat)`*, for any type* `S`*.*

6. (8 points) Briefly explain the term *aliasing*. Give one reason why it is a good thing and one reason why it is bad.

Answer: Aliasing can happen in any language with a pointers and a heap. It occurs when two different variables both refer to the same heap cell — or, more generally, when a single heap cell is accessible to a program via multiple "paths" of pointers. It is both *a good thing and a bad thing: good, because it is the basis of shared state and so fundamental to mainstream OO programming; but also bad, because it makes reasoning about programs more difficult. For example, executing the assignments* `x := 5; y := 6` *leaves* `x` *set to* `5` unless `x` *and* `y` *are aliases for the same cell.*

7. (24 points) Recall the *preservation* theorem for the STLC with references. In formal Coq notation it looks like this:

```
Theorem preservation : forall ST t t' T st st',
  has_type empty ST t T ->
  store_well_typed empty ST st ->
  t / st ==> t' / st' ->
  exists ST',
    (extends ST' ST /\
     has_type empty ST' t' T /\
     store_well_typed empty ST' st').
```

Informally, it looks like this:

> *Theorem (Preservation):* If `empty; ST |- t : T` with `ST |- st`, and `t` in store `st` takes a step to `t'` in store `st'`, then there exists some store typing `ST'` that extends `ST` and for which `empty; ST' |- t' : T` and `ST' |- st'`.

(a) Briefly explain why the extra (compared to preservation for the pure STLC) refinement "`exists ST'`..." is needed here.

Answer: Because reducing a term of the form `ref v` *allocates a new location* `l` *and yields* `l` *as the result of reduction, but* `l` *is not in the domain of* `ST` *and hence not typable under* `ST`*.*

(b) The proof of this theorem relies on some subsidiary lemmas:

```
Lemma store_weakening : forall Gamma ST ST' t T,
  extends ST' ST ->
  has_type Gamma ST t T ->
  has_type Gamma ST' t T.

Lemma store_well_typed_snoc : forall ST st t1 T1,
  store_well_typed ST st ->
  has_type empty ST t1 T1 ->
  store_well_typed (snoc ST T1) (snoc st t1).

Lemma assign_pres_store_typing : forall ST st l t,
  l < length st ->
  store_well_typed ST st ->
  has_type empty ST t (store_ty_lookup l ST) ->
  store_well_typed ST (replace l t st).

Lemma substitution_preserves_typing : forall Gamma ST x s S t T,
  has_type empty ST s S ->
  has_type (extend Gamma x S) ST t T ->
  has_type Gamma ST (subst x s t) T.
```

Suppose we carry out a proof of preservation by induction on the given typing derivation. In which cases of the proof are the above lemmas used?

Match names of lemmas to proof cases by drawing a line from from each lemma to each proof case that uses it.

	`T_Abs`
`store_weakening`	
	`T_App`
`store_well_typed_snoc`	
	`T_Ref`
`assign_pres_store_typing`	
	`T_Deref`
`substitution_preserves_typing`	
	`T_Assign`

Answer:

- `store_weakening` is used in the `T_App` and `T_Assign` cases.
- `store_well_typed_snoc` is used in the `T_Ref` case.
- `assign_pres_store_typing` is used in the `T_Assign` case.
- `substitution_preserves_typing` is used in the `T_App` case.

Grading scheme: 2 points for each correct line drawn, -1 point for each incorrect line

(c) Here is the beginning of the `T_Ref` case of the proof. Complete the case.

Theorem (Preservation): If `empty; ST |- t : T` with `ST |- st`, and `t` in store `st` takes a step to `t'` in store `st'`, then there exists some store typing `ST'` that extends `ST` and for which `empty; ST' |- t' : T` and `ST' |- st'`.

Proof: By induction on the given derivation of `empty; ST |- t : T`.

- *...cases for other rules...*
- If the last rule in the derivation is `T_Ref`, then `t = ref t1` for some `t1` and, moreover, `empty; ST |- t1 : T1` for some `T1`, with `T = Ref T1`.

  Answer: There are two cases to consider, one for each rule that could have been used to show that `ref t1` takes a step to `t'`.

  – Suppose `ref t1` takes a step by `ST_Ref`, with `t1` stepping to `t1'` and new store `st'`. Then by the IH there is some store typing `ST'` such that `ST'` extends `ST`, `st'` is well typed with respect to `ST'`, and `empty; ST' |- t1' : T1`. Hence, `empty; ST' |- ref t1' : Ref T1` by `T_Ref`.

  – Suppose `ref t1` takes a step by `ST_RefValue`. Then `t1` is a value, `st' = snoc st t1`, and `t'` is the length of `st` – i.e., the location of `t1` in `st'`. If we choose `ST' = snoc ST T1`, we obtain

    ∗ `ST'` extends `ST` by construction

    ∗ `st'` is well typed with respect to `ST'` by the `store_well_typed_snoc` lemma, and

    ∗ `empty; ST' |- l : Ref T1` by `T_Loc`,

    as required.

Grading scheme:

- *Correct case analysis of step: 2 points*
- *ST_Ref case: 4 points*
- *ST_RefValue case: 4 points*

Subtyping

8. (8 points) Recall the simply-typed lambda calculus extended with products and subtyping (reproduced on page 15).

The subtyping rule for products

```
S1 <: T1    S2 <: T2
--------------------                        (S_Prod)
   S1*S2 <: T1*T2
```

intuitively corresponds to the "depth" subtyping rule for records. Extending the analogy, we might consider adding a "permutation" rule

```
   --------------                           (S_ProdP)
   T1*T2 <: T2*T1
```

for products.

Is this a good idea? Briefly explain why or why not.

Answer: No, since it will break preservation: `(true,unit).1` *has type* `Unit` *according to this rule, but reduces to* `true`*, which does not have type* `Unit`*.*

Grading scheme: 1 point for "no," remaining points for clarity of explanation why.

9. (15 points) The preservation and progress theorems about the STLC with subtyping (page 15) depend on a number of technical lemmas, including the following one, which describes the possible "shapes" of types that are subtypes of an arrow type:

> *Lemma*: For all types `U`, `V1`, and `V2`, if `U <: V1 -> V2`, then there exist types `U1` and `U2` such that
>
> (a) `U = U1 -> U2`,
>
> (b) `V1 <: U1`, and
>
> (c) `U2 <: V2`.

The following purported proof of this lemma contains two significant mistakes. Explain what is wrong and how the proof should be corrected.

Proof: By induction on a derivation of `U <: V1 -> V2`.

- The last rule in the derivation cannot be S_Prod or S_Top since `V1 -> V2` is not a product type or Top.
- If the last rule in the derivation is S_Arrow, all the desired facts follow directly from the form of the rule.
- Suppose the last rule in the derivation is S_Trans. Then, from the form of the rule, there is some type `U'` with `U <: U'` and `U' <: V1 -> V2`. We must show that `U' = U1' -> U2'`, with `V1 <: U1'` and `U2' <: V2`; this follows from the induction hypothesis.

Answer:

- The case for S_Refl is omitted. In that case `U = V1 -> V2` and we have `V1 <: V1` and `V2 <: V2` by S_Refl.
- The S_Trans case is incomplete: it does not show anything about `U`. Once we know that `U <: U1' -> U2'`, we must apply the IH again to conclude that `U = U1 -> U2` where `U1' <: U1` and `U2 <: U2'`. Since we also know `V1 <: U1'` and `U2' <: V2`, by two applications of S_Trans we conclude that `V1 <: U1` and `U2 <: V2`. This finally gives us what we wanted to show.

Grading scheme: 5 points for the Refl part, 10 for the Trans part.

For Reference...

IMP programs

Here are the key definitions for the syntax and big-step semantics of IMP programs:

```
Inductive aexp : Type :=
  | ANum : nat -> aexp
  | AId : id -> aexp
  | APlus : aexp -> aexp -> aexp
  | AMinus : aexp -> aexp -> aexp
  | AMult : aexp -> aexp -> aexp.

Inductive bexp : Type :=
  | BTrue : bexp
  | BFalse : bexp
  | BEq : aexp -> aexp -> bexp
  | BLe : aexp -> aexp -> bexp
  | BNot : bexp -> bexp
  | BAnd : bexp -> bexp -> bexp.

Inductive com : Type :=
  | CSkip : com
  | CAss : id -> aexp -> com
  | CSeq : com -> com -> com
  | CIf : bexp -> com -> com -> com
  | CWhile : bexp -> com -> com.

Notation "'SKIP'" :=
  CSkip.
Notation "l '::=' a" :=
  (CAss l a) (at level 60).
Notation "c1 ; c2" :=
  (CSeq c1 c2) (at level 80, right associativity).
Notation "'WHILE' b 'DO' c 'END'" :=
  (CWhile b c) (at level 80, right associativity).
Notation "'IFB' e1 'THEN' e2 'ELSE' e3 'FI'" :=
  (CIf e1 e2 e3) (at level 80, right associativity).
```

```
           ----------------                         (E_Skip)
           SKIP / st  ⇓  st

           aeval st a1 = n
   --------------------------------                 (E_Ass)
   l := a1 / st  ⇓  (update st l n)

           c1 / st  ⇓  st'
          c2 / st'  ⇓  st''
         -------------------                        (E_Seq)
         c1;c2 / st  ⇓  st''

          beval st b1 = true
           c1 / st  ⇓  st'
-------------------------------------            (E_IfTrue)
IF b1 THEN c1 ELSE c2 FI / st  ⇓  st'

         beval st b1 = false
           c2 / st  ⇓  st'
-------------------------------------           (E_IfFalse)
IF b1 THEN c1 ELSE c2 FI / st  ⇓  st'

         beval st b1 = false
    ------------------------------             (E_WhileEnd)
    WHILE b1 DO c1 END / st  ⇓  st

          beval st b1 = true
           c1 / st  ⇓  st'
  WHILE b1 DO c1 END / st'  ⇓  st''
  ---------------------------------           (E_WhileLoop)
    WHILE b1 DO c1 END / st  ⇓  st''
```

Hoare logic rules

$$\frac{}{\{\{\texttt{assn\_sub V a Q}\}\}\ \texttt{V := a}\ \{\{\texttt{Q}\}\}}\ \textsc{Hoare\_Asgn}$$

$$\frac{\{\{\texttt{P}'\}\}\ \texttt{c}\ \{\{\texttt{Q}'\}\} \qquad P \longrightarrow P' \qquad Q' \longrightarrow Q}{\{\{\texttt{P}\}\}\ \texttt{c}\ \{\{\texttt{Q}\}\}}\ \textsc{Hoare\_Consequence}$$

$$\frac{\{\{\texttt{P}'\}\}\ \texttt{c}\ \{\{\texttt{Q}\}\} \qquad P \longrightarrow P'}{\{\{\texttt{P}\}\}\ \texttt{c}\ \{\{\texttt{Q}\}\}}\ \textsc{Hoare\_Pre} \qquad \frac{\{\{\texttt{P}\}\}\ \texttt{c}\ \{\{\texttt{Q}'\}\} \qquad Q' \longrightarrow Q}{\{\{\texttt{P}\}\}\ \texttt{c}\ \{\{\texttt{Q}\}\}}\ \textsc{Hoare\_Post}$$

$$\frac{}{\{\{\texttt{P}\}\}\ \texttt{SKIP}\ \{\{\texttt{P}\}\}}\ \textsc{Hoare\_Skip} \qquad \frac{\{\{\texttt{P}\}\}\ \texttt{c1}\ \{\{\texttt{Q}\}\} \qquad \{\{\texttt{Q}\}\}\ \texttt{c2}\ \{\{\texttt{R}\}\}}{\{\{\texttt{P}\}\}\ \texttt{c1 ; c2}\ \{\{\texttt{R}\}\}}\ \textsc{Hoare\_Seq}$$

$$\frac{\{\{\texttt{P} \wedge \texttt{b}\}\}\ \texttt{c1}\ \{\{\texttt{Q}\}\} \qquad \{\{\texttt{P} \wedge \sim\texttt{b}\}\}\ \texttt{c2}\ \{\{\texttt{Q}\}\}}{\{\{\texttt{P}\}\}\ \texttt{IFB b THEN c1 ELSE c2 FI}\ \{\{\texttt{Q}\}\}}\ \textsc{Hoare\_If}$$

$$\frac{\{\{\texttt{P} \wedge \texttt{b}\}\}\ \texttt{c}\ \{\{\texttt{P}\}\}}{\{\{\texttt{P}\}\}\ \texttt{WHILE b DO c END}\ \{\{\texttt{P} \wedge \sim\texttt{b}\}\}}\ \textsc{Hoare\_While}$$

STLC with references

(Some of the questions concerning STLC with references also use natural numbers and arithmetic operations; the syntax and semantics of these constants and operators is standard.)

Syntax

```
T ::=  Unit              t  ::= x                         v ::=
     | T -> T                 | t t                           | unit
     | Ref T                  | \x:T. t                       | \x:T. t
                              | unit                          | loc n
                              | ref t
                              | !t
                              | t := t
                              | loc n
```

Operational semantics

```
                     value v2
      ------------------------------------              (ST_AppAbs)
      (\a:T.t12) v2 / st ==> [v2/a]t12 / st

               t1 / st ==> t1' / st'
            ---------------------------                 (ST_App1)
            t1 t2 / st ==> t1' t2 / st'

        value v1     t2 / st ==> t2' / st'
        ----------------------------------              (ST_App2)
           v1 t2 / st ==> v1 t2' / st'

         --------------------------------               (ST_RefValue)
         ref v1 / st ==> loc |st| / st,v1

               t1 / st ==> t1' / st'
          -----------------------------                 (ST_Ref)
          ref t1 / st ==> ref t1' / st'

                     l < |st|
        ----------------------------------              (ST_DerefLoc)
        !(loc l) / st ==> lookup l st / st

               t1 / st ==> t1' / st'
              -----------------------                   (ST_Deref)
              !t1 / st ==> !t1' / st'
```

```
                 l < |st|
---------------------------------------------          (ST_Assign)
loc l := v2 / st ==> unit / (replace l v2 st)

        t1 / st ==> t1' / st'
 -----------------------------------                   (ST_Assign1)
 t1 := t2 / st ==> t1' := t2 / st'

        t2 / st ==> t2' / st'
 -----------------------------------                   (ST_Assign2)
 v1 := t2 / st ==> v1 := t2' / st'
```

Typing

```
                   Gamma x = T
                ------------------                          (T_Var)
                Gamma; ST |- x : T

          Gamma, x:T11; ST |- t12 : T12
        ----------------------------------                  (T_Abs)
        Gamma; ST |- \x:T11.t12 : T11->T12

            Gamma; ST |- t1 : T11->T12
              Gamma; ST |- t2 : T11
            --------------------------                      (T_App)
             Gamma; ST |- t1 t2 : T12

             ------------------------                      (T_Unit)
             Gamma; ST |- unit : Unit

                     l < |ST|
      --------------------------------------                (T_Loc)
      Gamma; ST |- loc l : Ref (lookup l ST)

               Gamma; ST |- t1 : T1
           ----------------------------                     (T_Ref)
           Gamma; ST |- ref t1 : Ref T1

            Gamma; ST |- t1 : Ref T11
            -------------------------                     (T_Deref)
             Gamma; ST |- !t1 : T11

            Gamma; ST |- t1 : Ref T11
              Gamma; ST |- t2 : T11
           ----------------------------                    (T_Assign)
           Gamma; ST |- t1 := t2 : Unit
```

STLC with products and subtyping

Syntax

```
  T ::=  Top                t  ::= x                           v ::= \x:T. t
       | T -> T                  | t t                              | (v,v)
       | T * T                   | \x:T. t
                                 | (t,t)
                                 | t.fst
                                 | t.snd
```

Operational semantics

```
                           ---------------------------                (ST_AppAbs)
                           (\a:T.t12) v2 ==> [v2/a]t12

                                 t1 ==> t1'
                              ----------------                          (ST_App1)
                              t1 t2 ==> t1' t2

                                 t2 ==> t2'
                              ----------------                          (ST_App2)
                              v1 t2 ==> v1 t2'

                                 t1 ==> t1'
                            --------------------                       (ST_Pair1)
                            (t1,t2) ==> (t1',t2)

                                 t2 ==> t2'
                            --------------------                       (ST_Pair2)
                            (v1,t2) ==> (v1,t2')

                                 t1 ==> t1'
                             ------------------                         (ST_Fst1)
                             t1.fst ==> t1'.fst

                             ------------------                      (ST_FstPair)
                             (v1,v2).fst ==> v1

                                 t1 ==> t1'
                             ------------------                         (ST_Snd1)
                             t1.snd ==> t1'.snd

                             ------------------                      (ST_SndPair)
                             (v1,v2).snd ==> v2
```

Subtyping

```
                ------                                  (S_Refl)
                T <: T

           S <: U    U <: T
           ----------------                            (S_Trans)
                S <: T

               --------                                  (S_Top)
               S <: Top

         T1 <: S1    S2 <: T2
         --------------------                          (S_Arrow)
           S1->S2 <: T1->T2

         S1 <: T1    S2 <: T2
         --------------------                           (S_Prod)
            S1*S2 <: T1*T2
```

Typing

```
                     Gamma x = T
                    --------------                          (T_Var)
                    Gamma |- x : T

             Gamma , x:T11 |- t12 : T12
            ----------------------------                    (T_Abs)
            Gamma |- \x:T11.t12 : T11->T12

               Gamma |- t1 : T11->T12
                 Gamma |- t2 : T11
               ----------------------                       (T_App)
                Gamma |- t1 t2 : T12

         Gamma |- t1 : T1       Gamma |- t2 : T2
         ---------------------------------------           (T_Pair)
                 Gamma |- (t1,t2) : T1*T2

                  Gamma |- t1 : T11*T12
                  ---------------------                     (T_Fst)
                  Gamma |- t1.fst : T11

                  Gamma |- t1 : T11*T12
                  ---------------------                     (T_Snd)
                  Gamma |- t1.snd : T12

                Gamma |- t : S     S <: T
                -------------------------                   (T_Sub)
                      Gamma |- t : T
```