CIS 500 — Software Foundations

Final Exam

May 9, 2011

Name or WPE-I number: ________________________________

Scores:

1	
2	
3	
4	
5	
6	
7	
8	
9	
Total (120 max)	

Hoare Logic

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

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.

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 }
```

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).

```
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')
```

References

5. (12 points) The next few problems concern the STLC extended with natural numbers and references (reproduced on page 15, 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?

(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?

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

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

provable? If so, what is it?

(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?

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

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.

(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

(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`.

  Fill in rest of case:

Subtyping

8. (8 points) Recall the simply-typed lambda calculus extended with products and subtyping (reproduced on page 17).

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.

9. (15 points) The preservation and progress theorems about the STLC with subtyping (page 17) 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.

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
```