CIS 500 — Software Foundations

Final Exam

(Advanced version)

December 18, 2014

Name: ______________________________

Pennkey (e.g. `sweirich`): ______________________________

Scores:

1		14
2		16
3		20
4		14
5		12
6		18
7		16
Total:		110

(14 points)

1. **Properties of Imp Relations**

The propositions below concern basic properties of the Imp language. For each proposition, indicate whether it is true or false by circling either True or False. For reference, the definition of Imp, its evaluation semantics, and program equivalence (`cequiv`) starts on page 13.

(a) The evaluation relation for Imp is deterministic.

True False

(b) The `cequiv` relation is symmetric.

True False

(c) The command `WHILE BFalse DO SKIP` is not equivalent to any other command.

True False

(d) There is an Imp command `c` that terminates for some input states and diverges for others.

True False

(e) If `cequiv c1 c2` then `cequiv (SKIP ;; c1) (SKIP ;; c2)`.

True False

(f) For all arithmetic expressions `a1` and `a2`, we can show

```
cequiv   (X ::= a1 ;; Y ::= a2)  (Y ::= a2 ;; X ::= a1)
```

True False

(g) If `SKIP / st || st'` then we know that `st` = `st'`.

True False

(16 points)

2. **Hoare Logic**

The following Imp program (slowly) computes `X + Y`, placing the answer into `Z`.

```
Z ::= Y;;
WHILE X <> 0 DO
   Z ::= Z + 1;;
   X ::= X - 1
END
```

Below, add appropriate annotations in the provided spaces. You will need to give the outermost pre- and post-conditions; these assertions should show that the program works as described above. Use informal notations for mathematical formulae and assertions, but be completely precise in the way you apply the Hoare rules — i.e., write out assertions in *exactly* the form given by the rules (rather than logically equivalent ones). The provided blanks have been constructed so that, if you work backwards from the end of the program, you should only need to use the rule of consequence in the places indicated with `->>`.

Mark the implication step(s) in your decoration (by circling the `->>`) that rely on the following fact. You may use other arithmetic facts silently.

- `forall a b c,  b <> 0 ->  (a - b) + 1 = a - (b - 1)`

The Hoare rules and the rules for well-formed decorated programs are provided on pages 15 and 16.

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

(20 points)

3. **Coq programming - Small-step semantics**

This problem refers to the Coq version of the small-step relation (`step`) for the Simply-typed Lambda Calculus with booleans, shown on page 18.

Because the `step` relation is deterministic, we can write a Coq function, called `next_step`, that computes what each term steps to (if any). The next page shows (part of) the definition of this function; you will need to complete the definition. Your implementation should satisfy the following correctness lemmas that state that it exactly corresponds to the `step` relation.

```
Lemma next_step_correct1 : forall t t' ,
       step t t' <-> next_step t = Some t'.
Lemma next_step_correct2 : forall t,
       normal_form step t <-> next_step t = None.
```

(a) Fill in the blanks for the following *examples* that demonstrate the evaluation of `next_step`. Your answers should be consistent with the correctness lemmas shown above.

Several of these examples make use of the following definition:

```
(* Identity function for booleans *)
Definition idB := tabs x TBool (tvar x).
```

The first one has been done for you.

```
Example ex0 : next_step (tapp idB ttrue) =

   _____________Some ttrue____________________.


Example ex1 : next_step ttrue =

   ___________________________________________.


Example ex2 : next_step (tapp ttrue tfalse) =

   ___________________________________________.


Example ex3 : next_step (tapp idB (tif ttrue tfalse ttrue)) =

   ___________________________________________.


Example ex4 : next_step (tif ttrue (tapp idB ttrue) tfalse) =

   ___________________________________________.
```

(b) Now complete the *implementation* of the `next_step` function. The first few cases of this implementation have been given for you.

Your code may use the following helper function in your answer.

```
(* Determine whether the given term is a value *)
Fixpoint is_value (t : tm) : bool :=
  match t with
  | tabs x T u => true
  | ttrue      => true
  | tfalse     => true
  | _          => false
  end.

(* Calculate the next (small-)step for this term, if one exists *)
Fixpoint next_step (t : tm) : option tm :=
    match t with
    | tif ttrue t2 t3 => Some t2
    | tif tfalse t2 t3 => Some t3
    | tif t1 t2 t3 => match next_step t1 with
        | Some t1' =>  Some (tif t1' t2 t3)
        | None  => None
        end

    (* Fill in remaining cases here, and on the next page if necessary *)
    |
```

(Extra space for the implementation of `next_step`, if necessary.)

(14 points)

4. **Inductive Definitions and Scoping**

Consider the following Coq definitions for a simple language of expressions with constants, variables, and `options`.

```
Definition id := nat.
Inductive tm : Type :=
| tnum : nat -> tm                        (* Constants 0, 1, 2, ...    *)
| tvar : id -> tm                         (* Variables X Y Z ...       *)
| tsome : tm -> tm                        (* Some t1                   *)
| tnone : tm                              (* None                      *)
| tmatch : tm -> tm -> id -> tm -> tm.   (* match t1 with
                                               | None   => t2
                                               | Some x => t3          *)
```

For example, we might encode the Coq expression

```
  match x with
  | None => 0
  | Some y => y
  end
```

as `tmatch (tvar X) (tnum 0) Y (tvar Y)`.
The `tmatch` construct follows the usual variable scoping rules. That is, in the expression `tmatch t1 t2 X t3` the variable `X` is *bound* in `t3`.

Note that a variable `X` *appears free* in a term `t` if there is an occurrence of `X` that is not bound by a corresponding `tmatch`. Complete the following Coq definition of `afi` as an inductively defined relation such that `afi X t` is provable if and only if `X` appears free in `t`. You may use the next page if you need more space.

```
Inductive afi : id -> tm -> Prop :=
```

(Space for the definition of `afi`, if necessary).

(12 points)

5. **Subtyping**

The subtyping rules for STLC extended with pairs and records are given on page 22 for your reference. The subtyping relations among a collection of types can be visualized compactly in picture form: we draw a graph so that `S <: T` iff we can get from `S` to `T` by following arrows in the graph (either directly or indirectly). For example, a picture for the types `Top*Top`, `A*Top`, `Top*(Top*Top)`, and `Top*(A*A)` would look like this (it happens to form a tree, but that is not necessary in general):

$$\begin{tikzcd}
& \texttt{Top*Top} & \\
\texttt{A*Top} \arrow[ur] & & \texttt{Top*(Top*Top)} \arrow[ul] \\
& & \texttt{Top*(A*A)} \arrow[u]
\end{tikzcd}$$

Suppose we have defined types `A` and `B` so that `A <: B`. Draw a picture for the following seven types.

```
{}
{ m : A }
{ m : B }
{ k : B }
{ m : Top }
{ m : A , k : B }
Top
{ m : A } -> Top
```

(18 points)

6. **Informal Proofs - Subtyping**

Give a detailed proof of the following inversion lemma for typing abstractions in STLC with subtyping (see the rules on page 21, you should the language with booleans and functions only, not the extension with product and record types).

Lemma (Typing Inversion for Abstractions): If $\Gamma \vdash$ `\x:S1.t2` $\in$ `T`, then there is a type `S2` such that Γ, `x:S1` $\vdash$ `t2` $\in$ `S2` and `S1 -> S2 <: T`.

(16 points)

7. **Informal Proofs — Progress with null**

In this problem we will extend STLC with `null`. Your job will be to state and prove a progress lemma for this extension. Although we will work with informal notation for this problem, the base language is the one specified by the Coq formalization, starting on page .

Informally, the only change we will make to the syntax of the language is to add `null`,

```
t ::=  x
    |  \x:T .t
    |  t1 t2
    |  true
    |  false
    |  if t1 then t2 else t3
    |  null
```

which is a new form of value.

```
                ----------  (v_null)
                value null
```

We also add one new typing rule, shown below, that allows the `null` value to have any type.

```
               --------------- (T_Null)
               Γ ⊢ null ∈ T
```

(The call-by-value operational semantics for this language is unchanged.)

The standard progress lemma doesn't hold for STLC with this extension. The problem is that a term could get stuck when trying to use a `null` value. For example the term `if null then t1 else t2` is well typed, but isn't a value and doesn't step. We say that stuck terms like this one throw *null pointer exceptions.* In this problem, we won't model those exceptions directly. Instead, we define a relation, called `npe`, that describes where they should occur. We can use this relation to modify the Progress lemma (as shown on the next page) so that it is true.

The `npe` relation (shown on the next page) characterizes those terms that should throw null pointer exceptions. This relation includes terms that try to use `null` as another sort of value. It does not include any other stuck terms.

Your task for this problem is to complete the proof of this revised progress lemma.

*Hint: This question is asking whether you understand the proof of the progress lemma for STLC. You will receive **half** credit for recreating the appropriate parts of the standard proof here, without the extension to null.*

The `npe` relation

```
  ----------------------------------  NPE_If
    npe (if null then t1 else t2)

              npe t1
  ----------------------------------  NPE_If1
    npe (if t1 then t2 else t3)


  ----------------------------------  NPE_App
            npe (null t1)


              npe t1
  ----------------------------------  NPE_App1
            npe (t1 t2)


              npe t2
  ----------------------------------  NPE_App2
            npe (t1 t2)
```

Lemma (Progress): For all `t` `T`, if $\emptyset \vdash$ `t` $\in$ `T` then either `t` is a value, throws a null pointer exception, or steps. (i.e. either `value t` or `npe t` holds or there exists some `t'` such that `t ==> t'`).
Proof: by induction on the derivation of $\emptyset \vdash$ `t` $\in$ `T`.

- The last rule of the derivation cannot be `T_Var`, since a variable is never well typed in an empty context.
- The `T_Null`, `T_True`, `T_False`, and `T_Abs` cases are trivial, because in each of these cases we know immediately that `t` is a value.
- If the last rule of the derivation was `T_If`, then ... (This case is omitted for brevity.)
- If the last rule of the derivation was `T_App`, then ... (Complete this case of the proof on the next page.)

(`T_App` case of the revised progress proof).

Formal definitions for Imp

Syntax

```
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).
```

Evaluation relation

```
Inductive ceval : com -> state -> state -> Prop :=
  | E_Skip : forall st,
      SKIP / st || st
  | E_Ass  : forall st a1 n X,
      aeval st a1 = n ->
      (X ::= a1) / st || (update st X n)
  | E_Seq : forall c1 c2 st st' st'',
      c1 / st  || st' ->
      c2 / st' || st'' ->
      (c1 ; c2) / st || st''
  | E_IfTrue : forall st st' b1 c1 c2,
      beval st b1 = true ->
      c1 / st || st' ->
      (IFB b1 THEN c1 ELSE c2 FI) / st || st'
  | E_IfFalse : forall st st' b1 c1 c2,
      beval st b1 = false ->
      c2 / st || st' ->
      (IFB b1 THEN c1 ELSE c2 FI) / st || st'
  | E_WhileEnd : forall b1 st c1,
      beval st b1 = false ->
      (WHILE b1 DO c1 END) / st || st
  | E_WhileLoop : forall st st' st'' b1 c1,
      beval st b1 = true ->
      c1 / st || st' ->
      (WHILE b1 DO c1 END) / st' || st'' ->
      (WHILE b1 DO c1 END) / st || st''

  where "c1 '/' st '||' st'" := (ceval c1 st st').
```

Program equivalence

```
Definition bequiv (b1 b2 : bexp) : Prop :=
  forall (st:state), beval st b1 = beval st b2.

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

Hoare triples

```
Definition hoare_triple (P:Assertion) (c:com) (Q:Assertion) : Prop :=
  forall st st', c / st || st' -> P st  -> Q st'.

Notation "{{ P }} c {{ Q }}" := (hoare_triple P c Q).
```

Implication on assertions

```
Definition assert_implies (P Q : Assertion) : Prop :=
  forall st, P st -> Q st.

Notation "P ->> Q" := (assert_implies P Q) (at level 80).
```

(ASCII `->>` is typeset as a hollow arrow in the rules below.)

Hoare logic rules

$$\frac{}{\{\!\{\,\texttt{assn\_sub X a}\ Q\,\}\!\}\ \texttt{X := a}\ \{\!\{\,Q\,\}\!\}}\quad(\texttt{hoare\_asgn})$$

$$\frac{}{\{\!\{\,P\,\}\!\}\ \texttt{SKIP}\ \{\!\{\,P\,\}\!\}}\quad(\texttt{hoare\_skip})$$

$$\frac{\begin{array}{c}\{\!\{\,P\,\}\!\}\ \texttt{c1}\ \{\!\{\,Q\,\}\!\}\\ \{\!\{\,Q\,\}\!\}\ \texttt{c2}\ \{\!\{\,R\,\}\!\}\end{array}}{\{\!\{\,P\,\}\!\}\ \texttt{c1; c2}\ \{\!\{\,R\,\}\!\}}\quad(\texttt{hoare\_seq})$$

$$\frac{\begin{array}{c}\{\!\{\,P \wedge b\,\}\!\}\ \texttt{c1}\ \{\!\{\,Q\,\}\!\}\\ \{\!\{\,P \wedge \sim b\,\}\!\}\ \texttt{c2}\ \{\!\{\,Q\,\}\!\}\end{array}}{\{\!\{\,P\,\}\!\}\ \texttt{IFB b THEN c1 ELSE c2 FI}\ \{\!\{\,Q\,\}\!\}}\quad(\texttt{hoare\_if})$$

$$\frac{\{\!\{\,P \wedge b\,\}\!\}\ \texttt{c}\ \{\!\{\,P\,\}\!\}}{\{\!\{\,P\,\}\!\}\ \texttt{WHILE b DO c END}\ \{\!\{\,P \wedge \sim b\,\}\!\}}\quad(\texttt{hoare\_while})$$

$$\frac{\begin{array}{c}\{\!\{\,P'\,\}\!\}\ \texttt{c}\ \{\!\{\,Q'\,\}\!\}\\ P \rightarrowtriangle P'\\ Q' \rightarrowtriangle Q\end{array}}{\{\!\{\,P\,\}\!\}\ \texttt{c}\ \{\!\{\,Q\,\}\!\}}\quad(\texttt{hoare\_consequence})$$

$$\frac{\begin{array}{c}\{\!\{\,P'\,\}\!\}\ \texttt{c}\ \{\!\{\,Q\,\}\!\}\\ P \rightarrowtriangle P'\end{array}}{\{\!\{\,P\,\}\!\}\ \texttt{c}\ \{\!\{\,Q\,\}\!\}}\quad(\texttt{hoare\_consequence\_pre})$$

$$\frac{\begin{array}{c}\{\!\{\,P\,\}\!\}\ \texttt{c}\ \{\!\{\,Q'\,\}\!\}\\ Q' \rightarrowtriangle Q\end{array}}{\{\!\{\,P\,\}\!\}\ \texttt{c}\ \{\!\{\,Q\,\}\!\}}\quad(\texttt{hoare\_consequence\_post})$$

Decorated programs

(a) `SKIP` is locally consistent if its precondition and postcondition are the same:

```
{{ P }}
SKIP
{{ P }}
```

(b) The sequential composition of `c1` and `c2` is locally consistent (with respect to assertions `P` and `R`) if `c1` is locally consistent (with respect to `P` and `Q`) and `c2` is locally consistent (with respect to `Q` and `R`):

```
{{ P }}
c1;
{{ Q }}
c2
{{ R }}
```

(c) An assignment is locally consistent if its precondition is the appropriate substitution of its postcondition:

```
{{ P [X |-> a] }}
X ::= a
{{ P }}
```

(d) A conditional is locally consistent (with respect to assertions `P` and `Q`) if the assertions at the top of its ”then” and ”else” branches are exactly `P /\ b` and `P /\ ~b` and if its ”then” branch is locally consistent (with respect to `P /\ b` and `Q`) and its ”else” branch is locally consistent (with respect to `P /\ ~b` and `Q`):

```
{{ P }}
IFB b THEN
  {{ P /\ b }}
  c1
  {{ Q }}
ELSE
  {{ P /\ ~b }}
  c2
  {{ Q }}
FI
{{ Q }}
```

(e) A while loop with precondition `P` is locally consistent if its postcondition is `P /\ ~b` and if the pre- and postconditions of its body are exactly `P /\ b` and `P`:

```
{{ P }}
WHILE b DO
  {{ P /\ b }}
  c1
  {{ P }}
END
{{ P /\ ~b }}
```

(f) A pair of assertions separated by `->>` is locally consistent if the first implies the second (in all states):

```
{{ P }} ->>
{{ P' }}
```

Coq formalization of STLC with booleans

Identifiers

```
Inductive id : Type :=
  Id : nat -> id.

Theorem eq_id_dec : forall id1 id2 : id, {id1 = id2} + {id1 <> id2}.

Definition x := (Id 0).
Definition y := (Id 1).
Definition z := (Id 2).
```

Types

```
Inductive ty : Type :=
  | TBool  : ty
  | TArrow : ty -> ty -> ty.
```

Terms

```
Inductive tm : Type :=
  | tvar   : id -> tm
  | tapp   : tm -> tm -> tm
  | tabs   : id -> ty -> tm -> tm
  | ttrue  : tm
  | tfalse : tm
  | tif    : tm -> tm -> tm -> tm.

Inductive value : tm -> Prop :=
  | v_abs : forall x T t,
      value (tabs x T t)
  | v_true :
      value ttrue
  | v_false :
      value tfalse.
```

Substitution

```
Reserved Notation "'[' x ':=' s ']' t" (at level 20).

Fixpoint subst (x:id) (s:tm) (t:tm) : tm :=
  match t with
  | tvar x' =>
      if eq_id_dec x x' then s else t
  | tabs x' T t1 =>
      tabs x' T (if eq_id_dec x x' then t1 else ([x:=s] t1))
  | tapp t1 t2 =>
      tapp ([x:=s] t1) ([x:=s] t2)
  | ttrue =>
      ttrue
  | tfalse =>
      tfalse
  | tif t1 t2 t3 =>
      tif ([x:=s] t1) ([x:=s] t2) ([x:=s] t3)
  end

where "'[' x ':=' s ']' t" := (subst x s t).
```

Reduction

```
Reserved Notation "t1 '==>' t2" (at level 40).

Inductive step : tm -> tm -> Prop :=
  | ST_AppAbs : forall x T t12 v2,
         value v2 ->
         (tapp (tabs x T t12) v2) ==> [x:=v2]t12
  | ST_App1 : forall t1 t1' t2,
         t1 ==> t1' ->
         tapp t1 t2 ==> tapp t1' t2
  | ST_App2 : forall v1 t2 t2',
         value v1 ->
         t2 ==> t2' ->
         tapp v1 t2 ==> tapp v1  t2'
  | ST_IfTrue : forall t1 t2,
      (tif ttrue t1 t2) ==> t1
  | ST_IfFalse : forall t1 t2,
      (tif tfalse t1 t2) ==> t2
  | ST_If : forall t1 t1' t2 t3,
      t1 ==> t1' ->
      (tif t1 t2 t3) ==> (tif t1' t2 t3)

where "t1 '==>' t2" := (step t1 t2).

Definition normal_form {X:Type} (R:relation X) (t:X) : Prop :=
  ~ exists t', R t t'.
```

Contexts

```
Definition partial_map (A:Type) := id -> option A.

Definition empty {A:Type} : partial_map A := (fun _ => None).
Definition extend {A:Type} (Gamma : partial_map A) (x:id) (T : A) :=
  fun x' => if eq_id_dec x x' then Some T else Gamma x'.
```

Typing Relation

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

Inductive has_type : context -> tm -> ty -> Prop :=
  | T_Var : forall Gamma x T,
      Gamma x = Some T ->
      Gamma |- tvar x \in T
  | T_Abs : forall Gamma x T11 T12 t12,
      extend Gamma x T11 |- t12 \in T12 ->
      Gamma |- tabs x T11 t12 \in TArrow T11 T12
  | T_App : forall T11 T12 Gamma t1 t2,
      Gamma |- t1 \in TArrow T11 T12 ->
      Gamma |- t2 \in T11 ->
      Gamma |- tapp t1 t2 \in T12
  | T_True : forall Gamma,
       Gamma |- ttrue \in TBool
  | T_False : forall Gamma,
       Gamma |- tfalse \in TBool
  | T_If : forall t1 t2 t3 T Gamma,
       Gamma |- t1 \in TBool ->
       Gamma |- t2 \in T ->
       Gamma |- t3 \in T ->
       Gamma |- tif t1 t2 t3 \in T

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

STLC with subtyping

Typing relation

```
                          Γ x = T
                         --------------                              (T_Var)
                         Γ ⊢ x ∈ T

                   Γ, x:T11 ⊢ t12 ∈ T12
                  ----------------------------                       (T_Abs)
                  Γ ⊢ \x:T11.t12 ∈ T11->T12

                     Γ ⊢ t1 ∈ T11->T12
                       Γ ⊢ t2 ∈ T11
                     ----------------------                          (T_App)
                      Γ ⊢ t1 t2 ∈ T12

                      --------------------                          (T_True)
                      Γ ⊢ true ∈ Bool

                     ---------------------                         (T_False)
                     Γ ⊢ false ∈ Bool

       Γ ⊢ t1 ∈ Bool    Γ ⊢ t2 ∈ T    Γ ⊢ t3 ∈ T
       --------------------------------------------------------       (T_If)
                 Γ ⊢ if t1 then t2 else t3 ∈ T

                    G ⊢ t ∈ S     S <: T
                  -------------------------                      (T_Sub)
                        Γ ⊢ t ∈ T
```

Subtyping relation

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

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

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

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

STLC subtyping, extended with pairs and records

Subtyping relation

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


                             n > m
                --------------------------------                 (S_RcdWidth)
                {i1:T1...in:Tn} <: {i1:T1...im:Tm}


                      S1 <: T1  ...  Sn <: Tn
                 ----------------------------------              (S_RcdDepth)
                 {i1:S1...in:Sn} <: {i1:T1...in:Tn}


        {i1:S1...in:Sn} is a permutation of {i1:T1...in:Tn}
        ---------------------------------------------------       (S_RcdPerm)
                 {i1:S1...in:Sn} <: {i1:T1...in:Tn}
```