CIS 500 — Software Foundations

Final

(Advanced version)

April 30, 2013

Name: ______________________________

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

Scores:

1		12
2		12
3		10
4		12
5		10
6		14
7		15
8		10
9		20
10		5
Total:		120

1. (12 points) Recall the definition of the `fold` function on lists (given in the appendix, page 1). Use it to define a function `sum_list_list` that, given a `list (list nat)`, returns the sum of all the `nat`s in it. E.g.

```
sum_list_list [[1,3],[5,6]] = 15
```

(Note that the first line we've given you starts with `Definition`, not `Fixpoint`: your solution should be non-recursive.)

```
Definition sum_list_list  (l : list (list nat)) : nat :=
```

2. (12 points) Complete the definition at the bottom of the page of an inductive proposition `count` that counts the number of elements of a list satisfying some predicate `P`. For example, if we define

```
Definition iszero (n : nat) : Prop :=
  n = 0.
```

then the propositions

```
count iszero [] 0
count iszero [1,2,3] 0
count iszero [0,1,2,3] 1
count iszero [1,0,0,2,3,0] 3
```

should all be provable, whereas the propositions

```
count iszero [1,2,3] 3
count iszero [0,0] 4
count iszero [] 1
```

should not be provable.

```
Inductive count {X : Type} (P : X -> Prop) : list X -> nat -> Prop :=
```

3. (10 points) For each of the types below, write a Coq expression that has that type. (For example, if the type given were `nat->nat`, you might write `fun x:nat, x+5`.) The definitions of `/\` and `\/` are given in the appendix (page 2) for reference.

(a) `forall X Y, (X -> Y) -> X -> list Y`

(b) `forall (X : Type), (X -> Prop) -> Prop`

(c) `forall (X Y : Prop), X -> Y -> (X \/ Y) /\ Y`

(d) `forall (X Y W Z : Prop), (X /\ Y) \/ (W /\ Z) -> (X \/ Z)`

4. (12 points) The following Imp program sets `R` to the sum of the initial values of `X` and `Y` *modulo* the initial value of `Z`:

```
R ::= X + Y;
WHILE Z <= R DO
  R ::= R - Z
END
```

Your task is to fill in assertions to make this a well-decorated program (see page 3 of the appendix), relative to an appropriate and post-condition. Please fill in annotations in the spaces provided below.

```
{{ True                                                    }} ->>

{{                                                         }}

R ::= X + Y;

{{                                                         }}

WHILE Z <= R DO

  {{                                                       }} ->>

  {{                                                       }}

  R ::= R - Z

  {{                                                       }}

END

{{                                                         }} ->>

{{ R = (X + Y) mod Z                                       }}
```

Finally, please mark (by circling the associated arrow `->>` and writing either 1 or 2 or both, as appropriate) any places where the following facts about modular arithmetic are needed to validate the correctness of the decorations:

```
(1)      a mod z = (a + z) mod z

(2)      a < z  ->  (a mod z) = a
```

5. (10 points) A *triangular number* is one that can be expressed as a sum of the form

$$1 + 2 + 3 + ... + n$$

for some n. In Coq notation:

```
Fixpoint tri (n : nat) : nat :=
  match n with
    0 => 0
  | S n' => n + (tri n')
  end.

Definition triangular (t : nat) : Prop :=
  exists n, t = tri n.
```

The following Imp program will terminate only when the initial value of the variable `T` is a triangular number.

```
{{ True }}
X ::= 0;
A ::= 0;
WHILE A <> T DO
  X ::= X + 1;
  A ::= A + X;
  {{ I }}
END
{{ triangular T }}
```

In the space below, write a suitable invariant for this loop — i.e., an assertion `I` such that, if we add `I` as the annotation at the end of the loop as shown, we can fill in annotations on the rest of the program to make it well decorated. (The spaces for these additional annotations are not shown, and you do not need to provide them.)

`I =`

6. (14 points) Consider the following two programs:

```
P1  =  WHILE b DO c END
P2  =  IFB b THEN c ELSE SKIP FI; P1
```

Give a careful informal proof that `P1` and `P2` are equivalent.

7. (15 points) Suppose L is some variant of STLC with the same syntax but in which the typing and single-step reduction rules have been changed in some way (by adding, removing, or changing rules). Consider the following schematic statement:

> "If the (A) theorem (B) for L, then changing L's (C) relation by (D) a rule might cause it to (E)."

If we substitute "preservation" for (A), "holds" for (B), "typing" for (C), "adding" for (D), and "fail" for (E), we obtain the statement "If the *preservation* theorem *holds* for L, then changing L's *typing* relation by *adding* a rule might cause it to *fail*," which happens to be true: adding a rule to the typing relation of a language like the STLC can sometimes destroy the preservation property.

Fill in the last column of the following table to indicate whether the statement obtained by similarly substituting the values in columns (A) to (E) is true or false. (We've done the first one for you.)

(A)	(B)	(C)	(D)	(E)	T/F
preservation	holds	typing	adding	fail	T
preservation	holds	single-step reduction	removing	fail	
preservation	holds	typing	removing	fail	
preservation	holds	single-step reduction	adding	fail	
progress	holds	typing	removing	fail	
progress	holds	single-step reduction	removing	fail	
progress	holds	typing	adding	fail	
progress	holds	single-step reduction	adding	fail	
preservation	fails	typing	removing	hold	
preservation	fails	single-step reduction	removing	hold	
preservation	fails	typing	adding	hold	
preservation	fails	single-step reduction	adding	hold	
progress	fails	typing	removing	hold	
progress	fails	single-step reduction	removing	hold	
progress	fails	typing	adding	hold	
progress	fails	single-step reduction	adding	hold	

8. (10 points) Consider the following propositions about typing in STLC. For each one, either give values for the existentially quantified variables to make the term typeable, or write "not typeable" and briefly explain why not.

(a) There exist types `T` and `T1` such that

```
empty |- (λx:T. if x then true else x) ∈ T1
```

```
T   =

T1  =
```

(b) For some types `T2` and `T1`,

```
y:T2 |- (λx:T1. x (y (y x) y)) ∈ Bool
```

```
T2  =
```

(c) For all types `T`, there exist types `T1` and `T2` such that

```
y:T2, x:T1 |- (y (x true) x) ∈ T
```

```
T1  =

T2  =
```

(d) For all types `T2` and `T1`, there exists a type `T` such that

```
y:T2 |- (λx:T1. if y x then true else x) ∈ T
```

```
T  =
```

(e) There exist types `T1` and `T2` such that

```
empty |- (λy:T2. λx:T1. if x then y (y (y x)) else x) ∈ Bool
```

```
T1  =

T2  =
```

9. (20 points) Your job in this problem is to fill in part of the proof of a lemma needed for the Preservation theorem in the STLC. (Note that the language we are considering here is the *standard SLTC*, without subtyping—see page 8 of the appendix.)

The lemma in question is the one stating that substitution preserves typing in STLC:

```
Theorem substitution_preserves_typing :
  forall (t t' : tm) (x : id) (Γ : context) (T T' : ty),
    Γ, x:T' |- t ∈ T ->
    empty |- t' ∈ T'  ->
    Γ |- [x := t'] t ∈ T.
```

On the next two pages, you will find an informal proof of this theorem, with two cases on the second page omitted. Fill in the argument for the missing cases.

Your may find it useful at some point(s) to invoke the following lemma (which you do not need to prove):

```
Lemma context_invariance : forall Γ Γ' t T,
     Γ |- t ∈ T  ->
     (forall x, appears_free_in x t -> Γ x = Γ' x) ->
     Γ' |- t ∈ T.
```

Proof: By induction on `t`. We have the following cases to consider:

- `t = true`. In this case, we have `[x := t'] true = true` by the definition of substitution. By the rule `T_True` the conclusion follows.

- `t = false`. Similar.

- `t = t1 t2`. In this case, we have

  ```
  [x := t'] (t1 t2)
      = ([x := t'] t1) ([x := t'] t2)
  ```

  by the definition of substitution. By inversion on `Gamma, x:T' |- t ∈ T` we know that there exists a `T1` such that

  ```
  Γ, x:T' |- t1 ∈ T1 -> T        (1)
  Γ, x:T' |- t2 ∈ T1             (2)
  ```

  In order to apply rule `T_App` to build the desired derivation, we have to show the following conditions:

  - `Γ |- [x := t'] t1 ∈ T1 -> T`. This follows from (1) by the IH.
  - `Γ |- [x := t'] t2 ∈ T1`. This follows from (2) by the IH.

- `t = if t1 then t2 else t3`. This case is similar to the one for application.

See next page for remaining cases...

- `t = y`.

  Fill in the rest...

- `t =` λ`y:T1.t1`.

  Fill in the rest...

10. (5 points) The STLC with records and subtyping is summarized in the appendix for reference (page 10). Indicate whether each of the following claims is true or false for this language.

(a) If `T <: S`, then `(T -> U) <: (S -> U)`.

T F

(b) The subtype relation contains an infinite descending chain — that is, there is an infinite sequence of types T_1, T_2, T_3, ... such that, for each i, we have $T_{i+1} <: T_i$ but not $T_i <: T_{i+1}$.

T F

(c) There is a record type `T` that is a supertype of every other record type (that is, `S <: T` for every record type `S`).

T F

(d) There is a type `T` that is a subtype of every other type (that is, `T <: S` for every type `S`).

T F

(e) There is a record type `T` that is a subtype of every other record type (that is, `T <: S` for every record type `S`).

T F

For Reference...

Some functions on lists

```
Fixpoint map {X Y:Type} (f:X->Y) (l:list X) : list Y :=
  match l with
  | []     => []
  | h :: t => (f h) :: (map f t)
  end.

Fixpoint fold {X Y:Type} (f:X->Y->Y) (y:Y) (l:list X) : Y :=
  match l with
  | []     => y
  | h :: t => f h (fold f y t)
  end.
```

Definitions of logical connectives in Coq

```
Inductive and (P Q : Prop) : Prop :=
  conj : P -> Q -> (and P Q).

Inductive or (P Q : Prop) : Prop :=
  | or_introl : P -> or P Q
  | or_intror : Q -> or P Q.

Notation "P /\ Q" := (and P Q) : type_scope.
Notation "P \/ Q" := (or P Q) : type_scope.
```

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 "X '::=' a" :=
  (CAss X 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)
                                  (at level 90, c at next level)
                                  : hoare_spec_scope.
```

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).
```

Hoare logic rules

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

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

$$\frac{\{\!\{\, P \,\}\!\}\ \texttt{c1}\ \{\!\{\, Q \,\}\!\} \qquad \{\!\{\, Q \,\}\!\}\ \texttt{c2}\ \{\!\{\, R \,\}\!\}}{\{\!\{\, P \,\}\!\}\ \texttt{c1; c2}\ \{\!\{\, R \,\}\!\}} \quad (\texttt{hoare\_seq})$$

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

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

$$\frac{\{\!\{\, P' \,\}\!\}\ \texttt{c}\ \{\!\{\, Q' \,\}\!\} \qquad P \twoheadrightarrow P' \qquad Q' \twoheadrightarrow Q}{\{\!\{\, P \,\}\!\}\ \texttt{c}\ \{\!\{\, Q \,\}\!\}} \quad (\texttt{hoare\_consequence})$$

Decorated programs

A decorated program consists of the program text interleaved with assertions. To check that a decorated program represents a valid proof, we check that each individual command is *locally* consistent with its accompanying assertions in the following sense:

- `SKIP` is locally consistent if its precondition and postcondition are the same:

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

- The sequential composition of commands `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 }}
```

- An assignment is locally consistent if its precondition is the appropriate substitution of its postcondition:

```
{{ P where a is substituted for X }}
X ::= a
{{ P }}
```

- 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 $P \land$ `b` and $P \land$ `~b` and if its "then" branch is locally consistent (with respect to $P \land$ `b` and Q) and its "else" branch is locally consistent (with respect to $P \land$ `~b` and Q):

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

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

```
{{ P }}
WHILE b DO
  {{ P ∧ b }}
  c1
  {{ P }}
END
{{ P ∧ ~b }}
```

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

```
{{ P }} ->>
{{ Q }}
```

STLC with booleans

Syntax

```
T ::= Bool                t  ::= x                      v ::= true
    | T -> T                  | t t                          | false
                              | \x:T. t                      | \x:T. t
                              | true
                              | false
                              | if t then t else t
```

Small-step operational semantics

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

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

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

        --------------------------------            (ST_IfTrue)
        (if true then t1 else t2) ==> t1

        ---------------------------------           (ST_IfFalse)
        (if false then t1 else t2) ==> t2

                t1 ==> t1'
  ----------------------------------------------------      (ST_If)
  (if t1 then t2 else t3) ==> (if t1' then t2 else t3)
```

Typing

```
                   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
```

Properties of STLC

```
Theorem preservation :
  forall t t' T,
    empty |- t ∈ T ->
    t ==> t' ->
    empty |- t' ∈ T.

Theorem progress :
  forall t T,
    empty |- t ∈ T ->
    value t \/ exists t', t ==> t'.
```

STLC with booleans, records and subtyping

Syntax

```
T ::= ...                  t  ::= ...                       v ::= ...
    | {i1:T1, ..., in:Tn}      | {i1=t1, ..., in=tn}            | {i1=v1, ..., in=vn}
    | Top                      | t.i
```

Small-step operational semantics

```
                          ...

                       ti ==> ti'
         -----------------------------------------              (ST_Rcd)
              {i1=v1, ..., im=vm, in=ti,  ...}
          ==> {i1=v1, ..., im=vm, in=ti', ...}

                       t1 ==> t1'
                    ----------------                          (ST_Proj1)
                     t1.i ==> t1'.i

                ---------------------------                 (ST_ProjRcd)
                 {..., i=vi, ...}.i => vi
```

Typing

```
                            ...

             Γ ⊢ t1 ∈ T1    ...    Γ ⊢ tn ∈ Tn
        -----------------------------------------------         (T_Rcd)
          Γ ⊢ {i1=t1,...,in=tn} ∈ {i1:T1,...,in:Tn}

                  Γ ⊢ t ∈ {...,i:T,...}
        -----------------------------------------------         (T_proj)
                      Γ ⊢ t.i ∈ T

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

Subtyping

```
                    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

                     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}
```