CIS 500 — Software Foundations

Final Exam

May 3, 2012

Answer key

This exam includes material on the Imp language and the simply-typed lambda calculus. Some of the key definitions are repeated, for easy reference, in the accompanying handout. The version of Imp we consider in this exam only has arithmetic expressions that reduce to numbers; you don't need to worry about lists.

1. (12 points) Recall the `fold` function in Coq:

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

Use the function `fold` to complete the definitions of the following Coq functions. Your solutions should not use `Fixpoint`.

(a) A function that sums all the elements of `l1 : list nat`; for example, if we apply the function you define to the list `[1,4,3]` it should return `8`.

```
Definition f1 (l1:list nat) : nat :=
```

```
fold plus l1 0
```

(b) A function that returns `true` iff at least one of the elements of `l2 : list bool` is `true`; if we apply the function you define to `[true,false,true]` it should return `true`, while if we apply it to `[false,false]` or `[]` it should return `false`.

```
Definition f2 (l2:list bool) : bool :=
```

```
fold orb l2 false
```

(c) A function that behaves the same as `map` (the standard definition of `map` is repeated on page 9, for reference).

```
Definition map' {X Y:Type} (f:X->Y) (l:list X) : (list Y) :=
```

```
fold (fun x xs => (f x)::xs) l3 nil
```

2. (12 points) Recall that Coq represents proofs internally as proof-objects — terms whose type is the proposition under consideration. For example, here is the proof object for the proposition `forall A B : Prop, A /\ B -> A`:

```
fun (A B : Prop) (H : A /\ B) =>
  match H with
  | conj HA HB => HA
  end.
```

Note the use of `match` to destruct the given proof object `H`.

Prove the following claims by providing proof objects as evidence. (For reference, the definitions of the logical connectives `and` and `or` are provided on page 10.)

(a) `forall A B C : Prop, (A /\ B -> C) -> A -> B -> C.`

```
fun (A B C : Prop) (H : A /\ B -> C) (HA : A) (HB : B) => H (conj A B HA HB).
```

(b) `forall A B C : Prop, (A -> B -> C) -> A /\ B -> C.`

```
fun (A B C : Prop) (H1 : A -> B -> C) (H2 : A /\ B) =>
match H2 with
| conj HA HB => H1 HA HB
end.
```

(c) `forall A B : Prop, A /\ B -> A \/ B`

```
fun (A B : Prop) (H : A /\ B) =>
match H with
| conj HA HB => or_introl A B HA
end.
```

(or something similar with `or_intror`)

Grading scheme: The first two arguments to tt and and `or` *were defined as explicit (not inferred) in Logic.v, but are implicit according to the standard definitions in the Coq library, which we have been using in the second half of the course. We gave full credit both to answers that included them (correctly!) and to answers that omitted them.*

3. (12 points) Each part of this question makes a general claim about program equivalences in Imp. For each one, indicate whether it is *true* or *false*. If it is false, give a counter-example. (For reference, the definition of program equivalence is provided on page 12.)

(a) For all commands `c` and boolean expressions `b`,

```
cequiv (WHILE b DO c END)
       (IF b THEN c ELSE SKIP FI; WHILE b DO c END)
```

True.

(b) For all arithmetic expressions `e1` and `e2`,

```
cequiv (X ::= e1; Y ::= e2)
       (Y ::= e2; X ::= e1)
```

False. If `e1` is the expression `Y` and `e2` is the expression `X`, then the commands in question are: `c1 = (X ::= Y; Y ::= X)`, and `c2 = (Y ::= X, X ::= Y)`. Consider a starting state `st` where `X` has value `1`, and `Y` has value `2`. `c1` then ends in a state with both `X` and `Y` equalling `2`, while `c2` goes to a state where they both equal `1`.

(c) For all boolean expressions `b1` and `b2` so that `bequiv b1 BTrue` and `bequiv b2 BFalse`,

```
cequiv (WHILE b1 DO (WHILE b2 DO SKIP END) END)
       (WHILE b2 DO (WHILE b1 DO SKIP END) END)
```

False. Starting from any state, the first command does not terminate, and the second always terminates in the same state.

4. (10 points) Indicate whether or not each of the following Hoare triples is valid by writing either "valid" or "invalid" next to it. Also, for those that are invalid, give a counter-example. (The definition of valid Hoare triples is given on page 13, for reference.)

(a)
```
{{ X = 0 }} Y ::= X {{ X = 0 }}
```

Answer: Valid. (Note that this is the case whether or not we assume `Y`$\sim =$ `X`*.)*

(b)
```
{{ True }} X ::= Y + 1 {{ X <> 0 }}
```

Answer: Valid.

(c)
```
{{ True }} X ::= Y - 1 {{ X <> 0 }}
```

Answer: Invalid: if `Y` *starts as 0 or 1, then* `X` *becomes 0.*

(d)
```
{{ True }} X ::= a {{ X = a }}
```

(Note that the variable `a` represents an arbitrary `aexp` – i.e., you should write "valid" only if the triple is valid for *every* `a`. If you give a counter-example, make sure it includes a specific arithmetic expression `a`.)

Answer: Invalid: consider `a = X + 1`

(e)
```
{{ True }}
WHILE X <> 0 DO
  Y ::= 1;
  X ::= X - 1;
END;
{{ Y = 1 }}
```

Answer: Invalid: consider the case where `X` *starts as 0, in which case* `Y`*'s value remains unchanged.*

Grading scheme: 2 points for each; for the invalid triples 1 point for the answer and 1 point for the counterexample

5. (20 points) The following Imp program calculates the integer division and remainder of two numbers `a` and `b`.

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

Note that we're using informal notations as usual in Imp examples, for example writing this...

```
WHILE (Y <= X)
```

...instead of this:

```
WHILE (BLe (AId Y) (AId X))
```

On the next page, add appropriate annotations to the program in the provided spaces to show that the Hoare triple given by the outermost pre- and post-conditions is valid. Use informal notations for mathematical formulae and assertions, but please be completely precise and pedantic 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 `=>`.

The Hoare logic rules and the guidelines for decorated programs are provided on page 13, for reference.

```
  {{ True }} =>
  {{ b * 0 + a = a /\ b = b }}
X ::= a;
  {{ b * 0 + X = a /\ b = b }}
Y ::= b;
  {{ Y * 0 + X = a /\ b = Y }}
Z ::= 0;
  {{ Y * Z + X = a /\ b = Y  }}
WHILE Y <= X DO
    {{ Y * Z + X = a /\ b = Y /\ Y<=X }} =>
    {{ Y * (Z + 1) + (X - Y) = a /\ b = Y }}
  X ::= X - Y;
    {{ Y * (Z + 1) + X = a /\ b = Y }}
  Z ::= Z + 1
    {{ Y * Z + X = a /\ b = Y }}
END;
  {{ Y * Z + X = a /\ b = Y /\ Y > X }}  =>
  {{ b * Z + X = a /\ b > X }}
```

Grading scheme: 14 points minor mistakes, 12 points logic mistake but correct invariant, 8-12 correct sketch of invariant wrong use of some rules, 4-8 some ideas, 0-4 not clear idea.

6. (20 points) Consider the simply typed lambda-calculus with booleans and the fixed-point operator `fix`. (You can find the syntax, typing rules, and small-step evaluation rules for this language beginning on page 16.) The *progress* theorem for this language can be stated as follows:

Theorem: If $\vdash$ `t` : `T`, then either `t` is a value or it can take a step.

Fill in the blanks in the following proof.

Proof: By induction on the given typing derivation.

- The last rule of the derivation cannot be `T_Var`, since a variable is never well typed in an empty context.
- The `T_True` and `T_False` cases are trivial, since in each of these cases we know immediately that `t` is a value.
- (The case where the last rule in the derivation is `T_If` is omitted for brevity.)
- If the last rule of the derivation is `T_Abs`, then `t` *is an abstraction and thus a value, by definition.*
- If the last rule of the derivation is `T_App`, then `t = t1 t2`*, and we know that* `t1` *and* `t2` *are also well typed in the empty context; in particular, there exists a type* `T2` *such that* $\vdash$ `t1` : `T2` $\to$ `T` *and* $\vdash$ `t2` : `T2`*. By the induction hypothesis, either* `t1` *is a value or it can take an evaluation step.*
  - *If* `t1` *is a value, we now consider* `t2`*, which by the other induction hypothesis must also either be a value or take an evaluation step.*
    - *Suppose* `t2` *is a value. Since* `t1` *is a value with an arrow type, it must be an abstraction; hence* `t1 t2` *can take a step by* `ST_AppAbs`*.*
    - *Otherwise,* `t2` *can take a step, and hence so can* `t1 t2` *by* `ST_App2`*.*
  - *If* `t1` *can take a step, then so can* `t1 t2` *by* `ST_App1`*.*
- If the last rule of the derivation is `T_Fix`, then `t = fix t1`*, and we know that* `t1` *is also well typed in the empty context; in particular, there exists a type* `T1` *such that* $\vdash$ `t1` : `T1` $\to$ `T1`*. By the induction hypothesis, either* `t1` *is a value or it can take an evaluation step.*
  - *If* `t1` *is a value, then since it has an arrow type, it must be an abstraction; hence* `fix t1` *can take a step by* `ST_FixAbs`*.*
  - *If* `t1` *can take a step, then so can* `fix t1` *by* `ST_Fix1`*.*

7. (20 points) In this exercise we investigate how the properties of the simply-typed lambda calculus with `fix` (the same language as in the previous problem) would change if we added new rules to the small-step reduction relation or to the typing relation. For each of the properties, either write "remains true" or else write "becomes false" *and give a counterexample.*

(a) Suppose we add the following new rule to the reduction relation:

```
----------------------------------          (ST_FunnyIfTrue)
(if true then t1 else t2) ==> true
```

Which of the following properties remain true in the presence of this rule? (Remember to give counterexamples for the ones that do not.)

- Determinism of `step` (`==>`) *Answer:* becomes false, counterexample:

  ```
  (if true then false else false) ==> false
  (if true then false else false) ==> true
  ```

- Progress *Answer:* remains true
- Preservation *Answer:* becomes false, counterexample:

  ```
  ⊢ (if true then (\x:Bool.x) else (\x:Bool.x)) : Bool -> Bool
  (if true then (\x:Bool.x) else (\x:Bool.x)) ==> true
  ```

  but it's not the case that

  ```
  ⊢ true : Bool -> Bool
  ```

(b) Suppose instead that we add the following two new rules to the reduction relation:

```
    value v
----------------                (ST_FunnyAppTrue)
true v ==> false

    value v
----------------                (ST_FunnyAppFalse)
false v ==> true
```

Which of the following properties remain true in the presence of these rules?

- Determinism of `step` (`==>`) *Answer:* remains true
- Progress *Answer:* remains true
- Preservation *Answer:* remains true

(c) Suppose instead that we add the following new rule to the typing relation:

```
Γ ⊢ t1 : Bool->Bool->Bool
    Γ ⊢ t2 : Bool
---------------------------          (T_FunnyApp)
   Γ ⊢ t1 t2 : Bool
```

Which of the following properties remain true in the presence of this rule?

- Determinism of `step` (`==>`) *Answer:* remains true
- Progress *Answer:* remains true
- Preservation *Answer:* becomes false, counterexample:

```
⊢ (\x:Bool. \y:Bool. x) true : Bool
(\x:Bool. \y:Bool. x) true ==> \y:Bool. true
```

  but it's not the case that

```
⊢ (\y:Bool. true) : Bool
```

(d) Suppose we add the following new rule to the typing relation:

```
                Γ ⊢ t1 : Bool
               --------------------                (T_FunnyFix)
               Γ ⊢ fix t1 : Bool
```

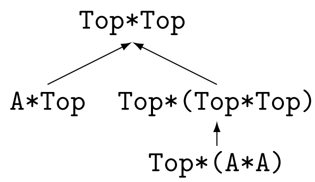
Which of the following properties remain true in the presence of this rule?

- Determinism of `step` (`==>`) *Answer:* remains true
- Progress *Answer:* becomes false, counterexample:

```
⊢ fix true : Bool
```

  but `fix true` is a stuck term.
- Preservation *Answer:* remains true

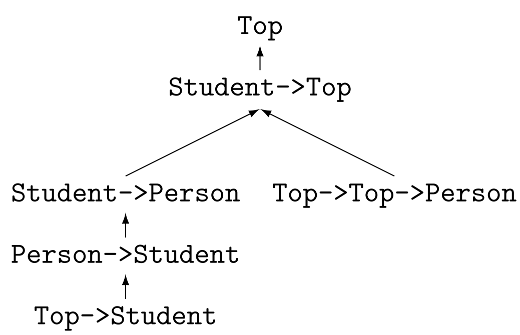
8. (14 points) The subtyping relations among a collection of types can be visualized compactly in picture form: we draw a graph so that `S <: T` iff we can get from `S` to `T` by following arrows in the graph (either directly or indirectly). For example, a picture for the types `Top*Top`, `A*Top`, `Top*(Top*Top)`, and `Top*(A*A)` would look like this (it actually happens to form a tree):



Suppose we have defined types `Student` and `Person` so that `Student <: Person`. Draw a picture for the following six types.

```
Student -> Person
Top
Student -> Top
Person -> Student
Top -> Student
Top -> Top -> Person
```

Answer:



For Reference...

The `map` function

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

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

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

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

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

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

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

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 $P \wedge$ `b` and $P \wedge \sim$`b` and if its "then" branch is locally consistent (with respect to $P \wedge$ `b` and Q) and its "else" branch is locally consistent (with respect to $P \wedge \sim$`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 $P \wedge \sim$`b` (where P is its precondition) and if the pre- and postconditions of its body are exactly $P \wedge$ `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 and `fix`

Syntax

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

                            t1 ==> t1'
                         ------------------                      (ST_Fix1)
                         fix t1 ==> fix t1'

                         F = \xf:T1.t2
                     -----------------------                    (ST_FixAbs)
                     fix F ==> [xf:=fix F]t2
```

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

            Γ ⊢ t1 : T1->T1
            -----------------                            (T_Fix)
            Γ ⊢ fix t1 : T1
```

STLC with pairs and subtyping (excerpt)

Types

```
T ::=  ...
     | Top
     | T -> T
     | T * T
```

Subtyping relation

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

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

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

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

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