Final Examination
CS 565
December 13, 2010

Name: _____

Instructions: Answer all questions in the space provided. Extra blank pages are provided in the back. Partial credit will be given where appropriate.

Maximum score: 235

Attained total score: _____

Question 1. (25 points)

1. (10 points)

   Let $A = \lambda a \lambda b \lambda c \lambda d \lambda e \lambda f \lambda g \lambda h \lambda i \lambda j \lambda k \lambda l \lambda m \lambda n \lambda o \lambda p \lambda q \lambda s \lambda t \lambda u \lambda v \lambda w \lambda x \lambda y \lambda z \lambda r.r(isthisafixedpointcombinator)$ and let $B = AAAAAAAAAAAAAAAAAAAAAAAAAA$ (26 applications of A).

   Is B a fixed-point combinator? Justify your answer.

2. (15 points) Let $f \circ g = \lambda x.f(g\ x)$ for all f, g (this is simply function composition). Let $\Delta = \lambda x.(x\ x)$.

   (a) (5 points) Show that $\Delta(f \circ \Delta) \equiv f(\Delta(f \circ\ \Delta))$

   (b) Let $Y_1 = \lambda f.\Delta(f \circ \Delta)$, $G = \lambda y.\lambda f.f(y\ f)$, and $Y_{n+1} = Y_n G$. Prove:

      i. (5 points) For all $x, Y_1\ x \equiv\ x(Y_1\ x)$

      ii. (5 points) $G\ Y_1 \equiv\ Y_1$.

Question 2. (30 points)

1. (5 points) Consider the following inductively defined family of propositions:

```
Inductive tree : Set :=
| leaf : tree
| node : tree -> tree -> tree.

Inductive p : tree -> nat -> Prop :=
| c1 : p leaf one
| c2 : forall t1 t2 n1 n2, p t1 n1 -> p t2 n2 -> p (node t1 t2) (plus n1 n2).
```

   Describe, in English, the conditions under which the proposition `p t n` is provable.

2. (5 points) Consider the following inductively defined family of propositions:

```
Inductive bar : nat -> Prop :=
| d : bar six
| e : forall n, bar (times n n)
| f : bar three -> bar five.
```

   For which `n` is the proposition `bar n` provable?

3. (5 points) Suppose we give Coq the following definition:

```
Inductive R : nat -> list nat -> Prop :=
| c1 : R 0 []
| c2 : forall n l, R n l -> R (S n) (n :: l)
| c3 : forall n l, R (S n) l -> R n l.
```

   Which of the following propositions are provable? (Write yes or no next to each one.)

   (a) `R 2 [1,0]`

   (b) `R 1 [1,2,1,0]`

   (c) `R 6 [3,2,1,0]`

4. (5 points) The concept of *composition of relations* can be defined as follows:

   Suppose Q and R are both relations on a set X. The composition of Q and R is the relation C such that, for all x and $z \in X$, $C\,x\,z \leftrightarrow \exists y.Q\,x\,y \wedge R\,y\,z$

   Write an inductive definition in Coq that expresses this concept.

5. (5 points) Give the definition of logical conjunction (`and`) as an inductive proposition in Coq.

6. (5 points) How would you explain the difference between a Coq proposition and a Coq type?

Question 3. (45 points)

1. (10 points)

   Give the weakest precondition for each of the following commands. (Use the informal notation for assertions rather than Coq notation, i.e., write `X = 5`, not `fun st => st X = 5`.)

   (a) `{ ? } X := 5 { X=6 }`

   (b)
   ```
   { ? }
   while IsList (X) do
     N := N + 1;
     X := Tail(X)
   end
   { X = [] ∧ N = length l }
   ```

   (c)
   ```
   { ? }
   while (Y <> 5) do
      Y := Y + 1
   { Y = 5 }
   ```

   (d)
   ```
   { ? }
   while not (X == 0) do
     Y := 1
   { Y = 5 }
   ```

2. (10 points) For each of the while programs below, a precondition and postcondition are provided. In the blank box before each loop, fill in an invariant that would allow us to annotate the rest of the program. Assume `X`, `Y` and `Z` are distinct program variables.

   (a)
   ```
   { True }
   X ::= n;
   Y ::= 1;
   {          }
   while (X <> 0)) do
      Y := Y ^ 2;
      X := X - 1
   end
   { Y = 2 ^ n }
   ```

(b)
```
{ True }
X :=  x;
Y :=  y;
Z := 0;
{ [          ] }
while (X <> 0) and  (Y <> 0) do
  X := X - 1;
  Y := Y - 1
  Z := Z + 1
end
{ Z = min (x, y) }
```

3. (10 points) Consider the following program:

```
{ N >= 0 }
j := 0;
s := 0;
while (j < N) do
 s := s + a[j];
 j := j + 1;
end
```

(a) (5 points) What is the strongest postcondition implied by its execution?

(b) (5 points) What is the appropriate loop invariant that would enable construction of a valid Hoare triple with respect to the asserted postcondition?

4. (15 points) Here is a Coq function that returns true if its argument contains only zeros:

```
Fixpoint allzero (xs:natlist) : bool :=
  match xs with
  | nil => true
  | h :: t => andb (beq_nat h 0) (allzero t) end.
```

   (a) (5 points) Write a corresponding program in Imp (a language with assignment, loops, conditionals, sequencing, and `nat` and `bool` base types) that performs the same function. You may assume the existence of an additional list datatype with operations `hd` and `tl` in Imp.

   (b) (10 points) Write down appropriate pre- and post-conditions along with a loop invariant that would allow you to prove the Imp program behaves the same as the Coq definition.

Question 4. (20 points)

1. (5 points) Suppose we remove the S_Arrow rule from the subtyping relation. Recall the S_Arrow rule is defined as:

$$\frac{T1 <: S1 \qquad S2 <: T2}{S1 \to S2 <: T1 \to T2}$$

and the subtype relation for the simply-typed lambda calculus is defined as:

$$\frac{}{T <: T} \qquad \text{(S\_Refl)}$$

$$\frac{S <: U \quad U <: T}{S <: T} \qquad \text{(S\_Trans)}$$

$$\frac{T1 <: S1 \qquad S2 <: T2}{S1 \to S2 <: T1 \to T2} \qquad \text{(S\_Arrow)}$$

Do progress and preservation continue to hold after this change, or does one (or do both) fail? If either fails, give a counterexample.

2. (5 points) Suppose we change the S_Arrow rule to:

$$\frac{T1 <: S1 \qquad T2 <: S2}{S1 \to S2 <: T1 \to T2}$$

Do progress and preservation continue to hold after this change, or does one (or do both) fail? If either fails, give a counterexample.

3. (5 points) Consider the simply-typed λ-calculus with records, tuples, and subtyping. Consider the addition of the following subtyping rule for tuples:

$$\frac{}{(\tau_1 \ * \ \tau_2 \ldots \ * \ \tau_k) <: (\tau_1 \ * \ \tau_2 \ * \ \tau_k \ * \ \tau_{k+1} \ * \ \ldots \tau_n)}$$

Prove the above rule is incorrect by writing a well-typed program that uses this rule, but leads to a runtime type error.

4. (5 points) Consider the following subtyping rule for function calls:

$$\frac{\begin{array}{c}\Gamma \vdash e_1 : \tau_1 \rightarrow \ \tau_2 \\ \Gamma \vdash e_2 : \tau_3 \\ \tau_3 <: \tau_1\end{array}}{\Gamma \vdash \ e_1(e_2) : \tau_2}$$

Prove that if we replaced this rule with one that was contravariant in in the argument position, a well-typed program can lead to a runtime type error. Your proof can be in the form of a counter-example.

Question 5. (25 points)

Recall that in the pure lambda-calculus determinacy of results is not affected by order of evaluation. Thus, sub-expressions can be evaluated in any order (including concurrently) without changing the meaning of the program. This property obviously does not hold in the presence of side-effects.

Consider the addition of a new primitive to the simply-typed lambda calculus augmented with assignment and pairs. A *witness* is an abstraction that orders side-effects. The grammar for this language is given below:

$$\begin{array}{rcl} e & := & x \mid i \mid \lambda x.e \mid e(e) \mid \texttt{let}\ x\ =\ e\ \texttt{in}\ e \mid e \otimes\ e \mid \pi_i(e) \\ & & \texttt{write}(e_1, e_2, e_3) \mid \texttt{read}(e, e') \mid \texttt{ref}(e) \mid \texttt{join}(e, e') \| \bullet \end{array}$$

The language is mostly standard. Pairs are written $e \otimes e$, projection on pairs is written $\pi_i(e)$ for $i = 1, 2$, etc.

Read and write operations on references, mutable locations, (which are created by `ref` (e)), take an extra argument. A read expression `read` (e_1,e_2) does not read the value of reference e_1 *until* it can fully evaluate the witness denoted by e_2. A write expression `write` (e_1, e_2, e_3) writes the value denoted by e_2 to the reference denoted by e_1 *after* evaluating the witness denoted by e_3. A read operation returns a pair containing the value read and a witness. Similarly, a write operation returns a witness after performing the write. The join expression joins two witnesses by waiting until it can evaluate both its arguments and then returns a new witness. The constant $\bullet$ denotes a witness value.

The following program returns $(2 \otimes \bullet)$ regardless of the evaluation order (e.g., call-by-value, or call-by-name):

```
let x = ref(1) in let w = write(x,2,•) in read(x,w)
```

The following program, on the other hand, returns either $(1 \otimes \bullet)$ or $(2 \otimes \bullet)$ depending upon evaluation order:

```
let x = ref(1) in let w = write(x,2,•) in read(x,•)
```

1. (10 points) What does the following program evaluate to under (a) call-by-value, and (b) call-by-name or lazy evaluation? (You may assume `let` is syntactic sugar for application.)

```
((λ x. read(x,•)) (let w = ref(1) in let y = write(x,2,•) in w))
```

2. (15 points) What does the following program evaluate to under call-by-value and call-by-name?

```
let x = ref(1) in
 let w = write (x,2,•) in
  let f = λ y.read(x,w) in
   let z = (f 0) ⊗ (f 0) in
    let w = write(x,3,join(π2(π1(z))π2(π2(z))in
      z
```

Question 6. (40 points)

Recall the polymorphic **let** rule of ML:

$$\frac{\Gamma \vdash e_1 : \sigma \qquad \Gamma, x : \sigma \vdash : e_2 : \tau}{\Gamma \vdash \textbf{let}\ x : \sigma = e_1 \textbf{in}\ e_2 : \tau}$$

1. (10 points) For each of the following ML terms, write a corresponding term in System F. You may assume the availability of tuples, projections, base types, and lists.

   (a) `fn x => x + let fun f (x) = x in f(2) end`

   (b) `let fun twice(f,x) = f(f(x)) in twice(fn y => y + 1,2)`

   (c) `let val f = fn x => x in f [f(0)] end`

2. (10 points) Give a one sentence explanation to describe each of the following terms.

   (a) $\lambda x : \forall t.t \to \forall u.t.x[\textbf{int}]3$

   (b) $(\lambda f : \forall t.t \to t\ \textbf{list}.f[\textbf{intlist}]f[\textbf{int}]0)\ (\Lambda t.\lambda x : t.\textbf{cons}(x, \textbf{nil}))$

3. (5 points) Give a one sentence explanation for why the following terms are not well-typed.

   (a) $\Lambda t.\lambda x : t.\Lambda t.x$

   (b) $\Lambda t.\lambda x : t.\textbf{cons}(x, \textbf{nil})[\textbf{int}][\textbf{intlist}]$

4. (5 points)

   Consider the following partially specified Coq definition:

```
Fixpoint fold f l b =
  match l with
  | nil => b
  | h :: t => f h (fold f t b)
  end.
```

   Rewrite the definition so that it represents a well-typed polymorphic function in Coq.

5. (10 points) For each of the following assertions, write down a type **T** that makes the assertion true, or else state that there exists no such type.

   (a) $\phi \vdash$ `(λp : T. p.fst (p.snd 42)) : T` $\rightarrow$ `A`

   (b) $\exists$ `U,` $\exists$ `V,` $\phi \vdash$ `(λ f : U. λ g : V. λ x : A. g (f x)) : T`

   (c) $\phi \vdash$ `fix (λ n : Nat. pred n) : T`

   (d) $\exists$ `S,` $\phi \vdash$ `λ x: T. x 42 x`

Question 7. (25 points)

The inversion lemma for subtyping arrow types states that every subtype of an arrow type is an arrow type. Formally, we can state this property as:

> **Lemma**: If `S <: T1 → T2`, then `S` has the form `S1 → S2`, with `T1 <: S1` and `S2 <: T2`.

Proof: By induction on subtyping derivations. The final rule in the derivation of `S <: T1 → T2` must be **S-Refl**, **S-Trans**, or **S-Arrow**. Fill in the proof for these three cases.

Case S-Refl: `S = T1 → T2`

Case S-Trans: `S <: U` `U <: T1 → T2`

Case S-Arrow: `S = S1 → S2` `T1 <: S1` `S2 <: T2`

For reference:

$$\frac{}{T <: T} \quad \text{(S\_Refl)}$$

$$\frac{S <: U \quad U <: T}{S <: T} \quad \text{(S\_Trans)}$$

$$\frac{T1 <: S1 \qquad S2 <: T2}{S1 \to S2 <: T1 \to T2} \quad \text{(S\_Arrow)}$$

Question 8. (25 points)

Consider the following Coq proof of the progress theorem for the simply-typed lambda calculus. Provide a short (no more than a sentence) answer for each of the questions given in the comments.

```
Theorem progress : forall t T,
     has_type empty t T ->
     value t  exists t', t --> t'.
Proof with eauto.
  (* How would you state this theorem in English *)
  intros t T Ht.
  remember (@empty ty) as Gamma.
  generalize dependent HeqGamma.
  (has_type_cases (induction Ht) Case); intros HeqGamma; subst.
  Case "T_Var".
    inversion H.
    (* What does the use of inversion H imply? *)
  Case "T_Abs".
    left...
    (* What does the application of the left tactic mean here? *)
  Case "T_App".
    right.
    destruct IHHt1; subst...
    (* What is the rationale for the following two subcases? *)
    SCase "t1 is a value".
      destruct IHHt2; subst...
      SSCase "t2 is a value".
        inversion H; subst; try (solve by inversion).
        (* What does the following achieve? *)
        exists (subst x t2 t12)...
      SSCase "t2 steps".
        (* When does this case apply? *)
        destruct H0 as [t2' Hstp]. exists (tm_app t1 t2')...
    SCase "t1 steps".
      (* When does this case apply? *)
      destruct H as [t1' Hstp]. exists (tm_app t1' t2)...
```

Reference

Imp

```
Inductive aexp : Type :=
  | ANum : nat -> 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.
```

Hoare Logic

```
------------------------------
{{Q[a/V]}} V::= a {{Q}}


      {{P'}} c {{Q'}}
P implies P' (in every state)
Q' implies Q (in every state)
------------------------------
      {{P}} c {{Q}}

--------------------
{{ P }} SKIP {{ P }}


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


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


         {{P /\ b}} c {{P}}
  -----------------------------------
  {{P}} WHILE b DO c END {{P /\ ~b}}
```

Relevant Type Rules for Simply-Typed Lambda Calculus

```
                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


 |- t1 : Bool    |- t2 : T    |- t3 : T
 --------------------------------------                (T_If)
      |- if t1 then t2 else t3 : T

              |- t1 : Nat
            ----------------                          (T_Succ)
            |- succ t1 : Not

              |- t1 : Nat
            ----------------                          (T_Pred)
            |- pred t1 : Not

              |- t1 : Nat
          -------------------                       (T_IsZero)
          |- iszero t1 : Bool


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

         Gamma |- t1 : T11*1T2
         ---------------------                        (T_Fst)
         Gamma |- t1.fst : T11

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

         Gamma |- t1 : T1->T1
        --------------------                          (T_Fix)
         Gamma |- fix t1 : T1
```

Relevant Subtyping Rules for the Simply-Typed Lambda Calculus

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

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

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

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

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

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

                      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}
```