Name:____

CSE505, Winter 2012, Final Examination
March 12, 2012

Rules:

- The exam is closed-book, closed-notes, except for one side of one 8.5x11in piece of paper.
- **Please stop promptly at 2:20.**
- You can rip apart the pages.
- There are **100 points** total, distributed **unevenly** among **7** questions, most of which have multiple parts.

Advice:

- Read questions carefully. Understand a question before you start writing.
- Write down thoughts and intermediate steps so you can get partial credit.
- The questions are not necessarily in order of difficulty. **Skip around.** In particular, make sure you get to all the problems.
- If you have questions, ask.
- Relax. You are here to learn.

For your reference (page 1 of 2):

$$\begin{array}{lcl} e & ::= & \lambda x.\ e \mid x \mid e\ e \mid c \mid \{l_1 = e_1, \ldots, l_n = e_n\} \mid e.l_i \mid \mathsf{fix}\ e \\ v & ::= & \lambda x.\ e \mid c \mid \{l_1 = v_1, \ldots, l_n = v_n\} \\ \tau & ::= & \mathsf{int} \mid \tau \to \tau \mid \{l_1 : \tau_1, \ldots, l_n : \tau_n\} \end{array}$$

$\boxed{e \to e'}$

$$\frac{}{(\lambda x.\ e)\ v \to e[v/x]} \qquad \frac{e_1 \to e_1'}{e_1\ e_2 \to e_1'\ e_2} \qquad \frac{e_2 \to e_2'}{v\ e_2 \to v\ e_2'} \qquad \frac{e \to e'}{\mathsf{fix}\ e \to \mathsf{fix}\ e'} \qquad \frac{}{\mathsf{fix}\ \lambda x.\ e \to e[(\mathsf{fix}\ \lambda x.\ e)/x]}$$

$$\frac{}{\{l_1 = v_1, \ldots, l_n = v_n\}.l_i \to v_i}$$

$$\frac{e_i \to e_i'}{\{l_1 = v_1, \ldots, l_{i-1} = v_{i-1}, l_i = e_i, \ldots, l_n = e_n\} \to \{l_1 = v_1, \ldots, l_{i-1} = v_{i-1}, l_i = e_i', \ldots, l_n = e_n\}}$$

$\boxed{\Gamma \vdash e : \tau}$

$$\frac{}{\Gamma \vdash c : \mathsf{int}} \qquad \frac{}{\Gamma \vdash x : \Gamma(x)} \qquad \frac{\Gamma, x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash \lambda x.\ e : \tau_1 \to \tau_2} \qquad \frac{\Gamma \vdash e_1 : \tau_2 \to \tau_1 \qquad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash e_1\ e_2 : \tau_1} \qquad \frac{\Gamma \vdash e : \tau \to \tau}{\Gamma \vdash \mathsf{fix}\ e : \tau}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \qquad \ldots \qquad \Gamma \vdash e_n : \tau_n \qquad \text{labels distinct}}{\Gamma \vdash \{l_1 = e_1, \ldots, l_n = e_n\} : \{l_1 : \tau_1, \ldots, l_n : \tau_n\}} \qquad \frac{\Gamma \vdash e : \{l_1 : \tau_1, \ldots, l_n : \tau_n\} \qquad 1 \le i \le n}{\Gamma \vdash e.l_i : \tau_i}$$

$$\frac{\Gamma \vdash e : \tau \qquad \tau \le \tau'}{\Gamma \vdash e : \tau'}$$

$\boxed{\tau_1 \le \tau_2}$

$$\frac{}{\{l_1{:}\tau_1, \ldots, l_{i-1}{:}\tau_{i-1}, l_i{:}\tau_i, \ldots, l_n{:}\tau_n\} \le \{l_1{:}\tau_1, \ldots, l_i{:}\tau_i, l_{i-1}{:}\tau_{i-1}, \ldots, l_n{:}\tau_n\}}$$

$$\frac{}{\{l_1{:}\tau_1, \ldots, l_n{:}\tau_n, l{:}\tau\} \le \{l_1{:}\tau_1, \ldots, l_n{:}\tau_n\}} \qquad \frac{\tau_i \le \tau_i'}{\{l_1{:}\tau_1, \ldots, l_i{:}\tau_i, \ldots, l_n{:}\tau_n\} \le \{l_1{:}\tau_1, \ldots, l_i{:}\tau_i', \ldots, l_n{:}\tau_n\}}$$

$$\frac{\tau_3 \le \tau_1 \qquad \tau_2 \le \tau_4}{\tau_1 \to \tau_2 \le \tau_3 \to \tau_4} \qquad \frac{}{\tau \le \tau} \qquad \frac{\tau_1 \le \tau_2 \qquad \tau_2 \le \tau_3}{\tau_1 \le \tau_3}$$

$$\begin{array}{lcl} e & ::= & c \mid x \mid \lambda x{:}\tau.\ e \mid e\ e \mid \Lambda\alpha.\ e \mid e[\tau] \\ \tau & ::= & \mathsf{int} \mid \tau \to \tau \mid \alpha \mid \forall\alpha.\tau \\ v & ::= & c \mid \lambda x{:}\tau.\ e \mid \Lambda\alpha.\ e \end{array} \qquad \begin{array}{lcl} \Gamma & ::= & \cdot \mid \Gamma, x{:}\tau \\ \Delta & ::= & \cdot \mid \Delta, \alpha \end{array}$$

$\boxed{e \to e'}$

$$\frac{e_1 \to e_1'}{e_1\ e_2 \to e_1'\ e_2} \qquad \frac{e_2 \to e_2'}{v\ e_2 \to v\ e_2'} \qquad \frac{e \to e'}{e[\tau] \to e'[\tau]} \qquad \frac{}{(\lambda x{:}\tau.\ e)\ v \to e[v/x]} \qquad \frac{}{(\Lambda\alpha.\ e)[\tau] \to e[\tau/\alpha]}$$

$\boxed{\Delta; \Gamma \vdash e : \tau}$

$$\frac{}{\Delta; \Gamma \vdash x : \Gamma(x)} \qquad \frac{}{\Delta; \Gamma \vdash c : \mathsf{int}} \qquad \frac{\Delta; \Gamma, x{:}\tau_1 \vdash e : \tau_2 \qquad \Delta \vdash \tau_1}{\Delta; \Gamma \vdash \lambda x{:}\tau_1.\ e : \tau_1 \to \tau_2} \qquad \frac{\Delta, \alpha; \Gamma \vdash e : \tau_1}{\Delta; \Gamma \vdash \Lambda\alpha.\ e : \forall\alpha.\tau_1}$$

$$\frac{\Delta; \Gamma \vdash e_1 : \tau_2 \to \tau_1 \quad \Delta; \Gamma \vdash e_2 : \tau_2}{\Delta; \Gamma \vdash e_1\ e_2 : \tau_1} \qquad \frac{\Delta; \Gamma \vdash e : \forall\alpha.\tau_1 \quad \Delta \vdash \tau_2}{\Delta; \Gamma \vdash e[\tau_2] : \tau_1[\tau_2/\alpha]}$$

For your reference (page 2 of 2):

$$\begin{array}{rcl} e & ::= & \ldots \mid \mathsf{A}(e) \mid \mathsf{B}(e) \mid (\mathsf{match}\ e\ \mathsf{with}\ \mathsf{A}x.\ e \mid \mathsf{B}x.\ e) \mid (e,e) \mid e.1 \mid e.2 \\ \tau & ::= & \ldots \mid \tau_1 + \tau_2 \mid \tau_1 * \tau_2 \\ v & ::= & \ldots \mid \mathsf{A}(v) \mid \mathsf{B}(v) \mid (v,v) \end{array}$$

$\boxed{e \to e'}$

$$\frac{e \to e'}{\mathsf{A}(e) \to \mathsf{A}(e')} \qquad \frac{e \to e'}{\mathsf{B}(e) \to \mathsf{B}(e')} \qquad \frac{e \to e'}{\mathsf{match}\ e\ \mathsf{with}\ \mathsf{A}x.\ e_1 \mid \mathsf{B}y.\ e_2 \to \mathsf{match}\ e'\ \mathsf{with}\ \mathsf{A}x.\ e_1 \mid \mathsf{B}y.\ e_2}$$

$$\frac{}{\mathsf{match}\ \mathsf{A}(v)\ \mathsf{with}\ \mathsf{A}x.\ e_1 \mid \mathsf{B}y.\ e_2 \to e_1[v/x]} \qquad \frac{}{\mathsf{match}\ \mathsf{B}(v)\ \mathsf{with}\ \mathsf{A}x.\ e_1 \mid \mathsf{B}y.\ e_2 \to e_2[v/y]}$$

$$\frac{e_1 \to e_1'}{(e_1,e_2) \to (e_1',e_2)} \qquad \frac{e_2 \to e_2'}{(v,e_2) \to (v,e_2')} \qquad \frac{e \to e'}{e.1 \to e'.1} \qquad \frac{e \to e'}{e.2 \to e'.2} \qquad \frac{}{(v_1,v_2).1 \to v_1} \qquad \frac{}{(v_1,v_2).2 \to v_2}$$

$\boxed{\Gamma \vdash e : \tau}$

$$\frac{\Gamma \vdash e : \tau_1 + \tau_2 \qquad \Gamma, x{:}\tau_1 \vdash e_1 : \tau \qquad \Gamma, y{:}\tau_2 \vdash e_2 : \tau}{\Gamma \vdash \mathsf{match}\ e\ \mathsf{with}\ \mathsf{A}x.\ e_1 \mid \mathsf{B}y.\ e_2 : \tau} \qquad \frac{\Gamma \vdash e : \tau_1}{\Gamma \vdash \mathsf{A}(e) : \tau_1 + \tau_2} \qquad \frac{\Gamma \vdash e : \tau_2}{\Gamma \vdash \mathsf{B}(e) : \tau_1 + \tau_2}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \qquad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash (e_1,e_2) : \tau_1 * \tau_2} \qquad \frac{\Gamma \vdash e : \tau_1 * \tau_2}{\Gamma \vdash e.1 : \tau_1} \qquad \frac{\Gamma \vdash e : \tau_1 * \tau_2}{\Gamma \vdash e.2 : \tau_2}$$

$$\begin{array}{rcl} e & ::= & x \mid \lambda x.\ e \mid e\ e \mid (e,e) \mid e.1 \mid e.2 \mid \mathsf{letcc}\ x.\ e \mid \mathsf{throw}\ e\ e \mid \mathsf{cont}\ E \\ v & ::= & \lambda x.\ e \mid (v,v) \mid \mathsf{cont}\ E \\ E & ::= & [\cdot] \mid E\ e \mid v\ E \mid (E,e) \mid (v,E) \mid E.1 \mid E.2 \mid \mathsf{throw}\ E\ e \mid \mathsf{throw}\ v\ E \end{array}$$

$$\frac{e \overset{\mathrm{p}}{\to} e'}{E[e] \to E[e']} \qquad \frac{}{E[\mathsf{letcc}\ x.\ e] \to E[(\lambda x.\ e)(\mathsf{cont}\ E)]} \qquad \frac{}{E[\mathsf{throw}\ (\mathsf{cont}\ E')\ v] \to E'[v]}$$

$$\frac{}{(\lambda x.\ e)\ v \overset{\mathrm{p}}{\to} e[v/x]} \qquad \frac{}{(v_1,v_2).1 \overset{\mathrm{p}}{\to} v_1} \qquad \frac{}{(v_1,v_2).2 \overset{\mathrm{p}}{\to} v_2}$$

Module Thread:

```
type t
val create : ('a -> 'b) -> 'a -> t
val join : t -> unit
```

Module Mutex:

```
type t
val create : unit -> t
val lock : t -> unit
val unlock : t -> unit
```

Futures:

```
type 'a promise
val future : (unit -> 'a) -> 'a promise
val force : 'a promise -> 'a
```

Module Event:

```
type 'a channel
type 'a event
val new_channel : unit -> 'a channel
val send : 'a channel -> 'a -> unit event
val receive : 'a channel -> 'a event
val choose : 'a event list -> 'a event
val wrap : 'a event -> ('a -> 'b) -> 'b event
val sync : 'a event -> 'a
```

Name:____

1. (**12** points) Suppose you design a new type system for Java to prevent null-pointer dereferences. However, due to poor design, your type system has the strange property that it does not accept any programs that are more than 12 lines long (it accepts some but not all shorter programs).

   Explain your answers *briefly.*

   (a) Is it definitely the case given just the information above that your type system is *sound* with respect to null-pointer dereferences?

   (b) Is it possible that your type system is *sound* with respect to null-pointer dereferences?

   (c) Is it definitely the case given just the information above that your type system is *complete* with respect to null-pointer dereferences?

   (d) Is it possible that your type system is *complete* with respect to null-pointer dereferences?

Name:____

2. (**20** points) Consider a typed-lambda calculus with subtyping. Assume:

   - The types are $\tau ::= \mathsf{int} \mid \tau \to \tau \mid \{l_1 : \tau_1, \ldots, l_n : \tau_n\}$.
   - The subtyping judgment has exactly the six inference rules on page 2 of this exam.

   For each of the following claims, if it is true, prove it. If it is false, provide a counterexample by giving a τ_1 and τ_2.

   (a) If $\tau_1 \neq \tau_2$ and there is a complete derivation of $\tau_1 \leq \tau_2$ that has no use of the rule for function subtyping, then there is a complete derivation of $\tau_1 \leq \tau_2$ that does not use the rule for reflexivity.

   (b) If $\tau_1 \neq \tau_2$ and there is a complete derivation of $\tau_1 \leq \tau_2$ that has no use of the rule for width subtyping on records, then there is a complete derivation of $\tau_1 \leq \tau_2$ that does not use the rule for reflexivity.

Name:____

3. (**13** points) This problem considers System F (extended with constants, addition, and multiplication).

   (a) Consider the type $\forall\alpha_1.\forall\alpha_2.\forall\alpha_3.(\alpha_1 \to \alpha_2) \to (\alpha_2 \to \alpha_3) \to \alpha_1 \to \alpha_3$.

   i. Give a (closed) term with this type.

   ii. Letting e_1 be your answer to part (i), what does e_1 [int] [int] [int] $(\lambda x : \mathsf{int}.\ x + 1)$ $(\lambda x{:}\mathsf{int}.\ x * 2)$ 17 evaluate to?

   iii. Is there a term e_2 with this type that is not equivalent to your answer to part (i)? If so, give such a term and what e_2 [int] [int] [int] $(\lambda x : \mathsf{int}.\ x + 1)$ $(\lambda x{:}\mathsf{int}.\ x * 2)$ 17 evaluates to. If not, explain briefly.

   (b) Consider the type $\forall\alpha_1.\forall\alpha_2.(\alpha_1 \to \alpha_1) \to (\alpha_1 \to \alpha_2) \to \alpha_1 \to \alpha_2$.

   i. Give a (closed) term with this type.

   ii. Letting e_1 be your answer to part (i), what does e_1 [int] [int] $(\lambda x : \mathsf{int}.\ x + 1)$ $(\lambda x{:}\mathsf{int}.\ x * 2)$ 17 evaluate to?

   iii. Is there a term e_2 with this type that is not equivalent to your answer to part (i)? If so, give such a term and what e_2 [int] [int] $(\lambda x : \mathsf{int}.\ x + 1)$ $(\lambda x{:}\mathsf{int}.\ x * 2)$ 17 evaluates to. If not, explain briefly.

Name:____

4. (**16** points) Consider this Caml code, which purposely uses unillimunating names. Assume `max3` is a provided library function that returns the maximum of three `int` arguments:

```
type t1 = A of int | B of int * t1 * t1
type t2 = C of string | D of string * t2 * t2
type t3 = E of t1 | F of t2
let foo x =
  match x with
    E e ->
      let rec f y =
        match y with
          A z -> z
        | B(z,a,b) -> max3 z (f a) (f b)
      in f e
  | F _ -> 505
```

   (a) `t3` describes a simple if slightly unusual data structure. Describe the data structure in 1–2 English sentences.

   (b) Give the type of `foo` and the type of the helper function `f`.

   (c) Describe what `foo` computes in 1–2 English sentences.

   (d) Write a Caml function `bar` that (1) is equivalent to `foo` (same type and behavior) and (2) never uses more than a small, constant amount of call-stack space when executed.

   (e) Your solution to part (d) needed a recursive helper function. What type does this helper function have?

Name:____

5. (**16** points) In this problem, you are given an implementation of a functional queue and you will use Concurrent ML to implement an imperative queue. Your solution should not use any mutable features (e.g., references). Assume the functional queue given to you has this interface:

```
type 'a queue
val empty_queue : 'a queue (* create a queue *)
val is_empty : 'a queue -> bool (* return true iff queue is empty *)
val enqueue : 'a queue -> 'a -> 'a queue (* add 'a to queue; return result *)
val peek : 'a queue -> 'a (* return element that was added to the queue the longest ago;
                             raises an exception if the queue is empty *)
val drop: 'a queue -> 'a queue (* result is like the argument but the element first added
                                  is no longer in the queue;
                                  raises an exception if the queue is empty *)
```

The queue you will implement will differ in that (1) it is imperative (a queue has state and enqueuing and dequeing update this state), (2) it supports only enqueue and dequeue operations, and (3) it never raises an exception; instead dequeue operations should *block* as necessary to wait for another thread to perform an enqueue. Also note you must use the type definition of `'a mqueue` given below. Provide definitions for `new_mqueue`, `m_enqueue`, and `m_dequeue`:

```
type 'a mqueue = 'a channel * 'a channel
val new_mqueue : unit -> 'a mqueue
val m_enqueue : 'a mqueue -> 'a -> unit
val m_dequeue : 'a mqueue -> 'a
```

Name:____

6. (**13** points) Assume a class-based OOP language like we discussed in class, with single inheritance and types (and subtypes) that correspond to classes (and subclasses). Assume `assert` raises an exception if its argument evaluates to false. Assume there are no threads. Consider this code:

```
class A {
   int x;
}
class B extends A {
   void m() { x = 7; }
}
class C extends B {
   void m() { x = 4; }
   void f() {
     m();
     assert(x==4); // (*)
   }
}
```

(a) Is it possible to make (only) a change to class B such that class B still type-checks but class C no longer type-checks? If so, explain how and if not, explain why not.

(b) Is it possible to make (only) a change to class B such that class B and class C type-check but there are programs where the assertion at the line marked `(*)` raises an exception? If so, explain how and if not, explain why not.

(c) Is it possible to create (only) a subclass D that extends class C such that class D type-checks but class C no longer type-checks? If so, explain how and if not, explain why not.

(d) Is it possible to create (only) a subclass D that extends class C such that class C and class D type-check but there are programs where the assertion at the line marked `(*)` raises an exception? If so, explain how and if not, explain why not.

Name:____

7. (**10** points) This problem considers static overloading and multimethods. Consider this code skeleton:

```
class A {}
class B extends A {}
class C extends B {}
class D {
   void m(B x, A y) { } // 1
   void m(A x, B y) { } // 2
   void m(A x, A y) { } // 3
   void main() {
     ...
   }
}
```

Replace the `...` with a few lines of code such that:

- If the language uses static overloading, then each of the three `m` methods is called exactly once. Indicate which line of code calls which method.
- If the language uses multimethods, then every method call is ambiguous because there is "no best match" (where we assume the "goodness" of a match is the number of subsumptions to immediate supertypes and fewer is better). Indicate for each call, which methods are tied for the best match.

Note the *same code* should satisfy both these requirements.