Name:_____

CSE505, Fall 2012, Final Examination
December 10, 2012

Rules:

- The exam is closed-book, closed-notes, except for one side of one 8.5x11in piece of paper.
- **Please stop promptly at 12:20.**
- You can rip apart the pages.
- There are **100 points** total, distributed **unevenly** among **6** 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):

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

$\boxed{e \to e'}$

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

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

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

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

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

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

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

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

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

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. (**20** points) Consider the language with subtyping (with the typing judgments $\Gamma \vdash e : \tau$ and $\tau_1 \leq \tau_2$) on the first reference page.

   (a) Prove this theorem: If $\cdot \vdash v : \tau_1 \to \tau_2$, then there is some x and e such that v is $\lambda x.\ e$. Hint: You need induction *and* a helper lemma about subtyping that you should state carefully and prove by a second induction argument.

   (b) Explain in 1–3 English sentences where in the proof of Type Safety we need the fact you proved in part (a).

   Solution:

   (a) The proof is by induction on the assumed typing derivation using this lemma, which we prove separately below: If $\tau \leq \tau_1 \to \tau_2$, then there is some τ_3 and τ_4 such that τ is $\tau_3 \to \tau_4$. We continue by cases on the bottommost rule in the assumed typing derivation:

   - The typing rules for constants and record expressions follow vacuously because they do not give a type of the form $\tau_1 \to \tau_2$.
   - The typing rule for variables follows vacuously because the context cannot be $\cdot$ (and/or because variables are not values).
   - The typing rules for fix-expressions, record projection, and function call follow vacuously because they do not apply to values.
   - The typing rule for lambda-expressions is a trivial case because it applies only to expressions of the form we need.
   - The typing rule for subsumption follows because:
     - If the result type is $\tau_1 \to \tau_2$, then by the right hypothesis and our lemma, τ is $\tau_3 \to \tau_4$ for some τ_3 and τ_4.
     - So by induction and the left hypothesis, v must be a lambda expression as we need.

   We now prove the lemma by induction on the derivation of $\tau \leq \tau_1 \to \tau_2$, with cases for the bottommost rule in the derivation:

   - The rules for width-, depth-, and permutation-subyping on records follow vacuously because the right type cannot be a function type.
   - The rule for function subtyping follows trivially because the type on the left is always a function type.
   - The rule for reflexivity follows immediately because τ must be $\tau_1 \to \tau_2$.
   - The rule for transitivity ensures there are shorter derivations of $\tau' \leq \tau_1 \to \tau_2$ and $\tau \leq \tau'$ for some τ'. So by induction on $\tau' \leq \tau_1 \to \tau_2$, we know τ' is $\tau_5 \to \tau_6$ for some τ_5 and τ_6. So plugging this into $\tau \leq \tau'$ gives $\tau \leq \tau_5 \to \tau_6$, so a second use of induction gives what we need.

   (b) This is one case of the Canonical Forms Lemma. We need this case for the Progress Theorem when arguing that a well-typed function call is never stuck: If e_1 and e_2 are both values in $e_1\ e_2$, then the lemma applied to e_1 ensures e_1 is a function, so we can perform a function call.

Name:_____

2. (**15** points) Each part of this problem considers a type in System F. For this type:

   - If no closed term has this type, explain briefly and informally why not.
   - If all closed terms of this type are equivalent, do all the following:
     - Give one such term.
     - Explain briefly and informally why all terms are equivalent.
     - Give a full typing derivation of the term you gave.
   - If there are two or more closed terms of this type that are not equivalent, then give two non-equivalent terms. (No explanation or typing derivations necessary.)

   (a) $\forall\alpha.(\alpha \to \mathsf{int})$

   (b) $\forall\alpha.(\mathsf{int} \to \alpha)$

   (c) $\forall\alpha.(\mathsf{int} \to \mathsf{int})$

   (d) $\forall\alpha.(\alpha \to \alpha)$

   Solution:

   (a) Many terms have this type, including $\Lambda\alpha.\ \lambda x : \alpha.\ 42$ and $\Lambda\alpha.\ \lambda x : \alpha.\ 43$.

   (b) No term has this type: There is no way to produce a closed term of type α without having a term of type α or some function that can produce an α.

   (c) Many terms have this type, including $\Lambda\alpha.\ \lambda x : \mathsf{int}.\ 42$ and $\Lambda\alpha.\ \lambda x : \mathsf{int}.\ 43$ as well as bodies that use x.

   (d) All closed terms of this type are equivalent to $\Lambda\alpha.\ \lambda x : \alpha.\ x$, the polymorphic identity function. System F has no non-termination or side-effects and the only value of type α a term can produce with this type is the one passed as an argument.

   Typing derivation:

$$\dfrac{\dfrac{\dfrac{}{\cdot, \alpha; \cdot, x : \alpha \vdash x : \alpha} \qquad \dfrac{}{\cdot, \alpha \vdash \alpha}}{\cdot, \alpha; \cdot \vdash \lambda x : \alpha.\ x : \alpha \to \alpha}}{\cdot; \cdot \vdash \Lambda\alpha.\ \lambda x : \alpha.\ x : \forall\alpha.(\alpha \to \alpha)}$$

Name:_____

3. (**15** points) Consider this OCaml code for appending two lists:

```
let rec append xs ys =
  match xs with
    [] -> ys
  | x::xs' -> x :: (append xs' ys)
```

(a) What is the type of `append` above?

(b) For a given call to `append` above, approximately how deep would the call-stack grow in terms of the function arguments?

(c) Use a helper function written in continuation-passing style to give a different version of `append` that uses a small constant amount of stack space.

(d) What is the type of the helper function you used in part (c)? (Note the type of `append` itself should still be the same as in part (a).)

(e) For *one* "bonus point" give yet another version of `append` that uses only a small constant amount of stack space. This version will need a helper function but should not use any function closures or higher-order functions.

Solution:

(a) `'a list -> 'a list -> 'a list`

(b) Its depth will be proportional to the length of `xs`. (The length of `ys` is irrelevant.)

(c)
```
let append xs ys =
  let rec f k xs =
    match xs with
      [] -> k ys
    | x::xs' -> f (fun zs -> k (x::zs)) xs'
  in f (fun x -> x) xs
```

(d) `f` has type `('a list -> 'b) -> 'a list -> 'b` (Note if your helper function takes `ys` as an argument that is fine, but then the type has another `'a list`. You might also have arguments in a different order.)

(e)
```
let append xs ys =
  let rec rev_append xs ys =
    match xs with
      [] -> ys
    | x::xs' -> rev_append xs' (x::ys)
  in rev_append (rev_append xs []) ys
```

Name:_____

4. (**12** points) Consider shared-memory multithreading with locks. Give pseudocode for a program that:

   - **Is** *data-race free*. That is, it has no data races on any execution.
   - Does **not** *follow consistent locking*. That is, there are executions with at least one variable that is accessed by multiple threads without there existing a lock that the threads always hold when accessing the variable.
   - Every shared variable is written to by at least one thread on at least some execution. (This avoids the trivial solution of a program that does reads but not writes.)

   Your pseudocode can look like the following wrong answer, where we have shared variables and locks and threads that run in parallel:

```
Shared data: x=0, y=0, lock m
Thread 1: if(x==0) { sync(m) { y++; }}
Thread 2: sync(m) { x++; } y++;
```

   Solution:
   There are, of course, an infinite number of correct solutions. Your instructor came up with this one, as did a couple students:

```
Shared data: x=0, y=0, lock m
Thread 1: x++; sync(m) { y++; }
Thread 2: sync(m) { if(y==1) x++; }
```

   We do not have consistent locking on `x`, but if Thread 2 accesses `x`, then this access must be ordered by happens-before after the access in Thread 1, so there is no data race on `x`. We have consistent locking on `y`, so no data races on `y`.

   There were several other nice solutions – a few are below. In all cases, let `x`, `y`, `z`, `m`, and `n` be shared (initialized to 0 or unheld) and anything else be thread local:

```
Thread 1: sync(m) { x++; } y = x;
Thread 2: sync(m) { if(x==0) while(true) { } }; y = x;

Thread 1: sync(m) { y++; } sync(n) { y++; }
Thread 2: sync(m) { sync(n) { y++; } }

Thread 1: while(!z) { sync(m) { if(!y) z = true; }} x++; sync(m) { z = false; }
Thread 2: while(!y) { sync(m) { if(!z) y = true; }} x++; sync(m) { y = false; }

Threads 1 and 2 the same:
int held = 0; while(!held) { sync(m) { if(y==0) { y=1; held=1; }}} ++x; sync(m) { y=0; }

Thread 1: x++; sync(m) { z++; } while(true) { sync(m) { if(z==2) break; } }; y++;
Thread 2: y++; sync(m) { z++; } while(true) { sync(m) { if(z==2) break; } }; x++;

Thread 1: sync(m) { x++; }
Thread 2: bool t; sync(m) { t = (x==0); } if(!t) x++;

Thread 1: sync(m) { x++; } tmp=x;
Thread 2: sync(m) { tmp=x; }
```

Name:_____

5. (**15** points)

   (a) In this problem, you will complete this Concurrent ML program by filling in the body for `make_thingy` so that it obeys the specification described below the code:

```
open Thread
open Event

let add (in1,in2) i = sync (send in1 i)

let halve (in1,in2) = sync (send in2 ())

type thingy = int channel * unit channel

let make_thingy k = (* your code here *)
```

   - `make_thingy` should return a new "thingy" that has an initial internal "state" of zero.
   - A call to `add` with a "thingy" adds `i` to the state of the "thingy".
   - A call to `halve` with a "thingy" divides the state of the "thingy" by two.
   - However, a call to `halve` must block unless the current state of the "thingy" is greater than or equal to `k` (the argument to `make_thingy` when the "thingy" was created).
   - Every time the state changes, "thingy" prints the current state as `Sum is x` where `x` is the new state. This includes when the state is initialized, so the first thing printed will be `Sum is 0`.
   - For full credit, do not use ML's references, but this is only worth about a point.

   (b) Give an ML signature (an `.mli` file) for the program in part (a). Your signature should allow the functions to be used as intended but not expose anything that allows the communication protocol or specification to be violated.

Solution:

(a)
```
let make_thingy k =
  let add_ch   = new_channel () in
  let halve_ch = new_channel () in
  let rec loop sum =
    print_string ("Sum is " ^ (string_of_int sum));
    if sum < k
    then loop ((sync (receive add_ch)) + sum)
    else loop (sync (choose [wrap (receive add_ch) (fun i -> i + sum);
                             wrap (receive halve_ch) (fun () -> sum / 2)]))
  in
  ignore(create loop 0);
  (add_ch,halve_ch)
```

(b)
```
type thingy

val add : thingy -> int -> unit
val halve : thingy -> unit
val make_thingy : int -> thingy
```

Name:_____

6. (**23** points) Consider a class-based OOP language with static types intended to prevent "no matching method" errors. Assume "subclassing is subtyping." For each of the following, indicate if it violates type safety. If it does violate type safety, give a short example that gets stuck. (If not, no explanation necessary.)

   In your examples, assume `B` is a subtype of `A` with a method `foo` not in `A`. You can fill in this template:

```
class A { }
class B extends A { void foo() }
class C { /* put stuff here */ }
class D extends C { /* put stuff here */ }
main program: /* put stuff here */
```

   (a) When overriding a method, we can change an argument type to be a supertype of what it was in the superclass' method.

   (b) When overriding a method, we can change an argument type to be a subtype of what it was in the superclass' method.

   (c) When overriding a method, we can change the result type to be a supertype of what it was in the superclass' method.

   (d) When overriding a method, we can change the result type to be a subtype of what it was in the superclass' method.

   (e) A subclass can change the type of a (mutable) field to be a subtype of what it was in the superclass. (This is changing the type of a field, not adding a second field.)

   (f) A subclass can change the type of a (mutable) field to be a supertype of what it was in the superclass. (This is changing the type of a field, not adding a second field.)

   Solution:

   (a) Sound (does not violate type safety)

   (b) Unsound

```
class C { void m(A a) { } }
class D extends C { void m(B b) { b.foo() }
main: ((C)new D()).m(new A())
```

   (c) Unsound

```
class C { B m() { return new B(); } }
class D extends C { A m() { return new A(); } }
main: ((C)new D()).m().foo();
```

   (d) Sound

   (e) Unsound

```
class C { A f; }
class D extends C { B f; }
main: D d = new D(); ((C)d).f = new A(); d.f.foo();
```

   (f) Unsound

```
class C { B f; }
class D extends C { A f; }
main: D d = new D(); d.f = new A(); ((C)d).f.foo();
```

Name:_____

Extra room for any problem where you might need it