Name:____

CSE 505, Fall 2008, Final Examination 11 December 2008

Please do not turn the page until everyone is ready.

Rules:

- The exam is closed-book, closed-note, except for one side of one 8.5x11in piece of paper.
- **Please stop promptly at 10:20.**
- You can rip apart the pages.
- There are **100 points** total, distributed among **6** questions.
- The questions 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.

Name:____

For your reference (page 1 of 2):

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

$\boxed{e \to e' \text{ and } \Gamma \vdash e : \tau \text{ and } \tau_1 \leq \tau_2}$

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

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

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

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

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

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

$$\frac{}{\{l_1{:}\tau_1, \ldots, l_n{:}\tau_n, l{:}\tau\} \leq \{l_1{:}\tau_1, \ldots, l_n{:}\tau_n\}}$$

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

$$\frac{\tau_i \leq \tau_i'}{\{l_1{:}\tau_1, \ldots, l_i{:}\tau_i, \ldots, l_n{:}\tau_n\} \leq \{l_1{:}\tau_1, \ldots, l_i{:}\tau_i', \ldots, l_n{:}\tau_n\}}$$

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

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

$\boxed{e \to e' \text{ and } \Delta; \Gamma \vdash e : \tau}$

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

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

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

Name:____

$$\begin{array}{rcl} e & ::= & \ldots \mid \mathsf{A}(e) \mid \mathsf{B}(e) \mid (\mathsf{match}\ e\ \mathsf{with}\ \mathsf{A}x.\ e \mid \mathsf{B}x.\ e) \mid \mathsf{roll}_\tau\ e \mid \mathsf{unroll}\ e \\ \tau & ::= & \ldots \mid \tau_1 + \tau_2 \mid \mu\alpha.\tau \\ v & ::= & \ldots \mid \mathsf{A}(v) \mid \mathsf{B}(v) \mid \mathsf{roll}_\tau\ v \end{array}$$

$\boxed{e \to e' \text{ and } \Delta;\Gamma \vdash e : \tau}$

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

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

$$\frac{}{\mathsf{unroll}\ (\mathsf{roll}_{\mu\alpha.\tau}\ v) \to v} \qquad \frac{e \to e'}{\mathsf{roll}_{\mu\alpha.\tau}\ e \to \mathsf{roll}_{\mu\alpha\tau}\ e'} \qquad \frac{e \to e'}{\mathsf{unroll}\ e \to \mathsf{unroll}\ e'}$$

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

$$\frac{\Delta;\Gamma \vdash e : \tau_1}{\Delta;\Gamma \vdash \mathsf{A}(e) : \tau_1 + \tau_2} \qquad \frac{\Delta;\Gamma \vdash e : \tau_2}{\Delta;\Gamma \vdash \mathsf{B}(e) : \tau_1 + \tau_2} \qquad \frac{\Delta;\Gamma \vdash e : \tau[(\mu\alpha.\tau)/\alpha]}{\Delta;\Gamma \vdash \mathsf{roll}_{\mu\alpha.\tau}\ e : \mu\alpha.\tau} \qquad \frac{\Delta;\Gamma \vdash e : \mu\alpha.\tau}{\Delta;\Gamma \vdash \mathsf{unroll}\ e : \tau[(\mu\alpha.\tau)/\alpha]}$$

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
```

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) Assume we have a typed programming language formally defined by a small-step operational semantics and a typing judgment. Assume the appropriate Preservation and Progress Theorems hold for this language. Consider each question below *separately* and **explain your answers briefly**.

   (a) Suppose we change the operational semantics by adding a new inference rule.
      i. Is it possible that the Preservation Theorem is now false?
      ii. Is it possible that the Progress Theorem is now false?

   (b) Suppose we change the type system by adding a new inference rule.
      i. Is it possible that the Preservation Theorem is now false?
      ii. Is it possible that the Progress Theorem is now false?

   (c) Suppose we change the operational semantics by replacing one of the inference rules with a rule that is just like it except it has some additional hypothesis.
      i. Is it possible that the Preservation Theorem is now false?
      ii. Is it possible that the Progress Theorem is now false?

   (d) Suppose we change the type system by replacing one of the inference rules with a rule that is just like it except it has some additional hypothesis.
      i. Is it possible that the Preservation Theorem is now false?
      ii. Is it possible that the Progress Theorem is now false?

   Solution:

   Note: Your instructor originally got part (b)i wrong. Also, grading was fairly lenient on the explanations, but they had to make some sense.

   For the benefit of full, precise explanations, let's state Preservation and Progress this way:

   - Preservation: If $(\underbrace{\cdot \vdash e : \tau}_{A}$ and $\underbrace{e \rightarrow e'}_{B})$, then $\underbrace{\cdot \vdash e' : \tau}_{C}$.
   - Progress: If $\underbrace{\cdot \vdash e : \tau}_{D}$, then e is a value or $\underbrace{\text{there exists } e' \text{such that } e \rightarrow e'}_{E}$.

   (a) i. Yes, the new rule might produce an ill-typed term. B is easier to satisfy, so A and B may no longer imply C.
      ii. No, more operational rules cannot make it harder to step. E is easier to satisfy so D still implies E.

   (b) i. Yes, the new rule might let some term type-check that can take a step to produce a term that doesn't type-check. A is easier to satisfy, so A and B may no longer imply C even though C is also easier to satisfy. As an example, suppose we have a rule give $(3 + 4) + ()$ type `int`. It can step to $7 + ()$, which does not type-check.
      ii. Yes, the new rules could allow a stuck term to type-check. D is easier to satisfy, so D may no longer imply E. For example, have a rule that gives $3 + ()$ type `int`.

   (c) i. No, the changed rule is now harder to use, but any term allowed after a step was also allowed before the change. B is harder to satisfy, so A and B still imply C.
      ii. Yes, the changed rule might no longer apply, which could cause some expression to be stuck that was not stuck in the old language. E is harder to satisfy so D may no longer imply E.

   (d) i. Yes, the changed rule is now harder to use, so some term that used to type-check might no longer type-check, so if that term can be produced by a well-typed expression taking a step, Preservation no longer holds. C is harder to satisfy, so A and B may no longer imply C even though A is also harder to satisfy.
      ii. No, only fewer terms type-check when we add hypotheses, so the theorem still applies to all the terms that type-check after the change. D is harder to satisfy, so D still implies E.

Name:____

2. (**15** points) Consider a typed λ-calculus with recursive types, sum types, pair types, int, and unit. Assume the language uses explicit roll and unroll coercions (not subtyping) for recursive types.

   (a) Define a type `t1` for lists where each list element contains either one int or a pair of ints. Use a sum type to tag list elements to distinguish these two possibilities.

   (b) Write a typed λ-calculus program of the form fix (λsum:`t1` $\rightarrow$ int. λlst:`t1`. _____) for adding up all the integers in a list of type `t1`. For example, a list holding the pair (2,3) and the integer 7 would have a sum of 12. Assume, of course, the expression language has addition.

   (c) Define a type `t2` for a list where list elements *alternate* between holding an int and a pair of ints, starting with an int. That is, the first, third, fifth, etc. elements are ints and the second, fourth, sixth elements, etc. are pairs of ints. However, the list may have any number of total elements (including 0 or an odd number). Do not use a sum type for list *elements* (although you still need sum types for the list itself).

   (d) Write a typed λ-calculus program of the form fix (λsum:`t2` $\rightarrow$ int. λlst:`t2`. _____) for adding up all the integers in a list of type `t2`. For example, a list holding the pair (2,3) and the integer 7 would have a sum of 12. Assume, of course, the expression language has addition.

Solution:

(a)

$$\mu\alpha.\mathsf{unit} + ((\mathsf{int} + (\mathsf{int} * \mathsf{int})) * \alpha)$$

(b)
```
fix (lambda sum : t1 -> int. lambda lst : t1.
       match unroll lst with
          A x -> 0
          B x -> (match x.1 with
                    A y -> y
                  | B y -> y.1 + y.2)
                 + sum x.2)
```

(c)

$$\mu\alpha.\mathsf{unit} + (\mathsf{int} * (\mathsf{unit} + ((\mathsf{int} * \mathsf{int}) * \alpha)))$$

(d)
```
fix (lambda sum : t1 -> int. lambda lst : t1.
       match unroll lst with
           A x -> 0
         | B x -> x.1 + (match x.2 with
                            A y -> 0
                          | B y -> (y.1.1 + y.1.2) + sum y.2))
```

Name:____

3. (**20** points) In this problem you will use CML to implement futures. Futures have this interface:

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

Recall a future runs in parallel and `force` blocks until the result is ready. It *is* permissable to call `force` with the same promise multiple times; if the result has already been computed it is simply returned immediately.

(a) Provide an implementation of the interface above in CML. Do not use mutation. Remember to include the definition of `type 'a promise`.

(b) Consider an extended interface with:

```
val force_any : 'a promise list -> 'a
```

Extend your implementation to provide this function. It should block only until one of the promises in the list is ready. If more than one is ready, it can return the result of any of the ready ones.

Solution:

(a)
```
type 'a promise = 'a channel

let future th =
  let c = new_channel() in
  let rec loop v = sync (send c v); loop v in
  ignore(create (fun () -> loop (th ())) ());
  c

let force p = sync (receive p)
```

(b)
```
let force_any ps = sync (choose (List.map receive ps))
```

Name:____

4. (**15** points) Consider a class-based OOP language like we did in class. Suppose it has abstract methods. Recall that a class that defines or inherits abstract methods cannot have instances unless all abstract methods are overridden.

   In this problem, we consider a bad idea for a new language feature: If class C inherits abstract method m, then C's definition can include `remove` m to mean instances of C do not have method m. This allows us to create instances of C unless there are other abstract methods that are neither overridden nor "removed."

   (a) Explain why this new feature is unsound if subclasses are subtypes. Give an example with a superclass, a subclass, and "client code" that makes an instance of the subclass. Your example should not require any concrete methods in the superclass.

   (b) Explain why this new feature is unsound even if subclasses that remove methods are not subtypes of their superclasses. Give an example with a superclass, a subclass, and "client code" that makes an instance of the subclass.

   Solution:

   (a) With subclasses as subtypes, a subclass needs to have every method that the superclass has. Even though the superclass does not give a definition for an abstract method, its type assumes the method exists. Hence this program type-checks but gets stuck:

   ```
   class B { abstract unit m(); }
   class C extends B { remove m }
   ((B)new C()).m();
   ```

   (b) When we type-check methods in the superclass, we assume all methods exist as provided by the superclass' type. This includes abstract methods, which thanks to late-binding will exist whenever a method in the superclass is called. But with our new feature, we are no longer required to define these methods, as in this example:

   ```
   class B { abstract unit m();
             unit n() { self.m(); }
           }
   class C extends B { remove m }
   (new C()).n();
   ```

Name:____

5. (**15** points) This problem considers the code below in a class-based OOP language. Assume classes `A`, `B`, `C`, and `D` have no subclasses except those shown.

```
class A {}
class B extends A {}
class C extends B {}
class D {
   int m(A x, A y) { return 1; }
   int m(A x, B y) { return 2; }
   int m(A x, C y) { return 3; }
   int m(B x, A y) { return 4; }
   int m(B x, B y) { return 5; }
   int m(B x, C y) { return 6; }
   int m(C x, A y) { return 7; }
   int m(C x, B y) { return 8; }
   int m(C x, C y) { return 9; }
   int foo(A a, C b) {
      return self.m(a,b);
   }
}
```

(a) Suppose our language has multimethods. What are the possible values that a call `(new D()).foo(e2,e3)` could return?

(b) Suppose our language has static overloading. What are the possible values that a call `(new D()).foo(e2,e3)` could return?

(c) How, if at all, do your answers change if subclasses of `D` might exist but subclasses of `C` do not?

(d) How, if at all, do your answers change if subclasses of `C` might exist but subclasses of `D` do not?

Solution:

(a) It could return 3, 6, or 9. At run-time, `a` could be bound to an instance of `A`, `B`, or `C` whereas `b` will be bound to an instance of `C`. Since method-lookup uses these run-time classes, the unique best match will depend on the value bound to `a`.

(b) It will return 3. The method-lookup is resolved using types `A` and `C` for the first and second arguments respectively.

(c) In this case, neither answer changes since the receiver is something with run-time class `D`. (The answer could be different if we knew only that the receiver had type `D`, which is actually what your instructor meant to ask.)

(d) In this case, neither answer changes. We have the same 9 methods named `m`. The new possibilities are `a` or `b` could be bound to a suclass of `C`. With multimethods, that will result in the answer 9 since this will require the fewest subsumptions. With static overloading, that will result in the answer 3 since the run-time classes remain irrelevant.

Name:______________________________________

6. (**15** points) This problem considers a typed language with support for bounded parametric polymorphism, i.e., types of the form $\forall\alpha < \tau_1.\ \tau_2$.

   (a) The System F typing rule for type application:

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

   is no longer appropriate because (1) e does not have a bounded polymorphic type and (2) τ_2 is not checked against a bound. Replace this typing rule with a more appropriate one.

   (b) Suppose our language has addition, records with mutable fields, and $e_1; e_2$ for sequencing. Then this term type-checks:

   $$\lambda x{:}\{l_1{:}\mathsf{int}, l_2{:}\mathsf{int}\}.\ ((x.l_1 := x.l_1 + x.l_2);\ x)$$

   Use bounded polymorphism to give a value v that has the same run-time behavior as this term but has a much more general type. Use the syntax $\Lambda\alpha < \tau.e$ to create a bounded polymorphic expression.

   (c) Using the v you defined in part (b), give a τ and v_2 such that $v\ [\tau]\ v_2$ type-checks (using the rule you defined in part (a) and the rule for function application). Choose a v_2 that takes advantage of the subtyping allowed by bounded polymorphism.

   (d) What is the type of the expression $(v\ [\tau]\ v_2)$ in your answer to part (c)? You do *not* need to write down a typing derivation.

Solution:

(a)

$$\frac{\Delta;\Gamma \vdash e : \forall\alpha < \tau_3.\tau_1 \quad \Delta \vdash \tau_2 < \tau_3}{\Delta;\Gamma \vdash e[\tau_2] : \tau_1[\tau_2/\alpha]}$$

(b)

$$\Lambda\alpha < \{l_1{:}\mathsf{int}, l_2{:}\mathsf{int}\}.\ \lambda x{:}\alpha.\ ((x.l_1 := x.l_1 + x.l_2);\ x)$$

(c) (One example)
Let τ be $\{l_1{:}\mathsf{int}, l_2{:}\mathsf{int}, l_3{:}\mathsf{int}\}$ and v_2 be $\{l_1 = 7, l_2 = 8, l_3 = 9\}$.

(d) (Answer depends on part (c))
$\{l_1{:}\mathsf{int}, l_2{:}\mathsf{int}, l_3{:}\mathsf{int}\}$