Name:____________________________________________________

CSE 505, Fall 2003, Final Examination
12 December 2003

Please do not turn the page until everyone is ready.

Rules:

• The exam is open-book, open-note, closed electronics.

• Please stop promptly at 12:20.

• You can rip apart the pages, but please write your name on each page.

• You can turn in other pieces of paper.

• There are six questions (all with subparts), worth equal amounts. The subparts are not necessarily worth equal amounts.

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 roughly in the order we covered the material, not necessarily order of difficulty. Skip around.

• If you have questions, ask.

• Relax. You are here to learn, not beat the mean.
1. Assume a typed lambda-calculus with recursive types, product types, and sum types. (You may choose the formulation of recursive types with subtyping or explicit roll/unroll. Just state your choice.)

(a) Give a type describing binary trees of integers. A binary tree of integers is either a leaf (which holds one integer) or a node (which holds one integer in addition to two binary (sub)trees of integers).

(b) Write a one-argument function that takes an integer and produces a leaf holding that integer.

(c) Give a full typing derivation showing that your answer to the previous part is a function from integers to binary trees of integers.

**Solution:**
Using roll and unroll:

(a) \( \mu \alpha . (\text{int} + (\text{int} \times \alpha \times \alpha)) \)

(b) \( \lambda x : \text{int} . \text{roll}_{\mu \alpha . (\text{int} + (\text{int} \times \alpha \times \alpha))} (\text{inl} x) \)

(c) 

\[
\begin{array}{c}
\vdash x : \text{int} \\
\vdash x \mapsto \text{inl} x : (\text{int} + (\mu \alpha . (\text{int} + (\text{int} \times \alpha \times \alpha))) \times (\mu \alpha . (\text{int} + (\text{int} \times \alpha \times \alpha)))) \\
\vdash x : \text{int} \mapsto \text{roll}_{\mu \alpha . (\text{int} + (\text{int} \times \alpha \times \alpha))} (\text{inl} x) : \mu \alpha . (\text{int} + (\text{int} \times \alpha \times \alpha))) \\
\vdash \lambda x : \text{int} . \text{roll}_{\mu \alpha . (\text{int} + (\text{int} \times \alpha \times \alpha))} (\text{inl} x) : \text{int} \rightarrow (\mu \alpha . (\text{int} + (\text{int} \times \alpha \times \alpha)))
\end{array}
\]
2. Assume a typed lambda-calculus with records, references, and subtyping. For each of the following, describe exactly the conditions under which the subtyping claim holds.

Example question: \( \{ l_1 : \tau_1, \ l_2 : \tau_2 \} \leq \{ l_1 : \tau_3, \ l_2 : \tau_4 \} \)

Example answer: “when \( \tau_1 \leq \tau_3 \) and \( \tau_2 \leq \tau_4 \)”

Your answer should be “fully reduced” in the sense that if you say \( \tau \leq \tau' \), then \( \tau \) or \( \tau' \) or both should be \( \tau_i \) for some number \( i \) where \( \tau_i \) appears in the question.

(a) \( \{ \{ l_1 : \tau_1, l_2 : \tau_2 \} \rightarrow \text{int} \} \leq \{ \{ l_1 : \tau_3, l_2 : \tau_4 \} \rightarrow \text{int} \) 

(b) \( \{ l_1 : (\tau_1 \text{ ref}) \} \leq \{ l_1 : \tau_2 \} \)

(c) \( (\tau_1 \rightarrow \tau_2) \rightarrow (\tau_3 \rightarrow \tau_4) \leq (\tau_5 \rightarrow \tau_6) \rightarrow (\tau_7 \rightarrow \tau_8) \)

(d) \( (\tau_1 \rightarrow \tau_2) \text{ ref} \leq (\tau_3 \rightarrow \tau_4) \text{ ref} \)

Solution:

(a) when \( \tau_3 \leq \tau_1 \) and \( \tau_4 \leq \tau_2 \)
(b) when \( \tau_2 \) has the form \( \tau_3 \text{ ref} \), \( \tau_3 \leq \tau_1 \), and \( \tau_1 \leq \tau_3 \)
(c) when \( \tau_1 \leq \tau_5 \), \( \tau_6 \leq \tau_2 \), \( \tau_7 \leq \tau_3 \), and \( \tau_4 \leq \tau_8 \)
(d) when \( \tau_1 \leq \tau_3 \), \( \tau_3 \leq \tau_1 \), \( \tau_2 \leq \tau_4 \), and \( \tau_4 \leq \tau_2 \)
3. Consider the following O'Caml code.

```ocaml
let catch_all1 t1 t2 = try t1 () with x -> t2 ()
let catch_all2 t1 t2 = try t1 () with x -> t2
```

(a) Under what conditions, if any, does using `catch_all1` raise an exception?
(b) Under what conditions, if any, does using `catch_all2` raise an exception?
(c) What type does O'Caml give `catch_all1`? (You can give your answer in O'Caml notation or System-F notation.)
(d) What type does O'Caml give `catch_all2`? (You can give your answer in O'Caml notation or System-F notation.)

**Solution:**

(a) when calling its first argument raises an exception and calling its second argument raises an exception
(b) never
(c) Caml: `(unit -> α) → (unit → α) → α`
(d) Caml: `(unit → α) → α → α`
4. Consider these definitions in a class-based OO language:

```java
class Thunk {
    abstract int apply();
}

class A {
    private int y;
    int g() { <<a hard-to-compute function using self.y>> }  // line 0
    unit set_y(int i) { self.y := i }
}

class B {
    int f1(int x, bool b) { if b then x else 0 }  // line 1
    int f2(Thunk x, bool b) { if b then x.apply() else 0 }

    unit f(A a, bool b) {
        print_int(self.f1(a.g(), b)); // line 2 (irrelevant until part (e))
        print_int(self.f1(a.g(), b)); // line 3 (identical to line 1)
    }
}
```

(a) Replace lines 1 and 3 with code that uses `f2` and not `f1`, but still prints the same number. You should declare a subclass of `Thunk` (outside of `class B`) and use this class (including on line 0). Your subclass should define a constructor that takes argument(s) and initializes fields appropriately. (Hint: Pass `a` to the constructor.)

(b) Compared to the original version, when is the change you made in part (a) faster and when is it slower?

(c) Change your subclass of `Thunk` so that the first time `apply` is called it stores the result it returns in private state. When called again, `apply` should return the stored result. Lines 1 and 3 should be the same as in part (a).

(d) Compared to the change in part (a), does the change in part (c) make line 1 faster or slower? Does it make line 3 faster or slower?

(e) Write a line 2 that makes the change in part (c) incorrect in the sense that `f` might print different output than in part (a).

The next page is blank.
(a) class T extends Thunk {
    A a;
    constructor(A x) { a := x; }
    int apply() { return a.g(); }
}

unit f(A a, bool b) {
    Thunk t = new T(a);
    print_int(self.f2(t,b));
    print_int(self.f2(t,b));
}

(b) It is faster when \( b \) is false (because we never execute \( a.g() \)) and (slightly) slower when \( b \) is true (because we build an extra object).

(c) class T extends Thunk {
    A a;
    bool done;
    int ans;
    constructor(A x) { a := x; done := false; }
    int apply() {
        if(not done) ans := a.g();
        ans;
    }
}

(d) It makes line 1 slightly slower (if \( b \) is true) because we store the result. It makes line 3 much faster (if \( b \) is true) because the result is stored.

(e) a.set_y(42);
5. Consider these definitions in a class-based OO language:

```java
class C1 {
    int g() { return 0; }
    int f() { return g(); }
}
class C2 extends C1 {
    int g() { return 1; }
    int m4(D2 x) { return x.f(); }
}
class D1 {
    private C1 x = new C1();
    int g() { return 0; }
    int f() { return x.f(); }
}
class D2 extends D1 {
    int g() { return 1; }
}

class Main {
    int m1(C1 x) { return x.f(); }
    int m2(C2 x) { return x.f(); }
    int m3(D1 x) { return x.f(); }
    int m4(D2 x) { return x.f(); }
}
```

Assume this is not the entire program, but the rest of the program does not declare subclasses of the classes above.

**Explain your answers:**

(a) True or false: Changing the body of `m1` to `return 0` produces an equivalent `m1`.
(b) True or false: Changing the body of `m2` to `return 1` produces an equivalent `m2`.
(c) True or false: Changing the body of `m3` to `return 0` produces an equivalent `m3`.
(d) True or false: Changing the body of `m4` to `return 1` produces an equivalent `m4`.
(e) How do your answers change if the rest of the program might declare subclasses of the classes above (excluding `Main`)?

**Solution:**

(a) false: If `m1` is passed an instance of `C2`, it will return 1.
(b) true: there are no subtypes of `C2`, so any call to `m2` will pass an instance of `C2`, and late-binding ensures the `f` method of a `C2` returns 1.
(c) true: Any call to `m3` will pass an instance of `D1` or `D2`. The `f` methods for both are the same: return the result of `C1`'s `f` method.
(d) false: same reason as previous question
(e) All claims become false because calls to `f` in `Main` could resolve to methods defined in subclasses we do not see above.
6. Consider a class-based OO language with this ill-advised addition: We can declare new classes with “class C restricts D by m.” If D is a class with a method named m, then this declaration creates a class C that inherits the fields and methods of D except C has no method m.

(a) Given class C restricts D by m, show that C should not be a subtype of D. Give an example expression that would type-check if C were a subtype of D but that would lead to a failed method-lookup, regardless of what members D has (other than m).

(b) Given class C restricts D by m, show that C may be a bad type, even without subtyping. Give an example class D and an expression e that would type-check such that evaluation of e would lead to a failed method-lookup. (Hint: D (and therefore C) can have methods other than m.)

(c) If we have interface I restricts J by m instead of class C restricts D by m, is it wrong to allow I ≤ J (or does the problem from part (a) no longer apply)? Explain.

(d) If we have interface I restricts J by m instead of class C restricts D by m, can I be a bad type even without subtyping (or does the problem from part (b) no longer apply)? Explain.

Solution:

(a) (D)(new C()).m() (cast for emphasis, not necessary)

(b) class D { unit n() { m(); } unit m() {} } and (new C()).n()

(c) Yes, it is still wrong. Part (a) is a typing problem and interfaces are types. Subtypes should have more fields and methods, not fewer.

(d) No, this is not a problem. Part (b) is a method-lookup problem, which has to do with classes not types. The superinterface only defines the presence of a method, not a method body that might depend on other methods.