Name:____

CSE P505, Winter 2009, Final Examination 19 March 2009

Rules:

- The exam is closed-book, closed-note, except for one two-sided 8.5x11in piece of paper.
- **Please stop promptly at 8:20.**
- You can rip apart the pages.
- There are **100 points** total, distributed **unevenly** among **7** 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):

$$\begin{array}{rcl} s & ::= & \mathsf{skip} \mid x := e \mid s; s \mid \mathsf{if}\ e\ s\ s \mid \mathsf{while}\ e\ s \\ e & ::= & i \mid x \mid e + e \mid e * e \\ (i & \in & \{\ldots, -2, -1, 0, 1, 2, \ldots\}) \\ (x & \in & \{\mathtt{x}_1, \mathtt{x}_2, \ldots, \mathtt{y}_1, \mathtt{y}_2, \ldots, \mathtt{z}_1, \mathtt{z}_2, \ldots, \ldots\}) \end{array}$$

$\boxed{H\ ;\ e\ \Downarrow\ i}$

$$\text{CONST}\ \frac{}{H\ ;\ c\ \Downarrow\ c} \qquad \text{VAR}\ \frac{}{H\ ;\ x\ \Downarrow\ H(x)} \qquad \text{ADD}\ \frac{H\ ;\ e_1\ \Downarrow\ c_1 \qquad H\ ;\ e_2\ \Downarrow\ c_2}{H\ ;\ e_1 + e_2\ \Downarrow\ c_1 + c_2} \qquad \text{MULT}\ \frac{H\ ;\ e_1\ \Downarrow\ c_1 \qquad H\ ;\ e_2\ \Downarrow\ c_2}{H\ ;\ e_1 * e_2\ \Downarrow\ c_1 * c_2}$$

$\boxed{H_1\ ;\ s\ \Downarrow\ H_2}$

$$\text{SKIP}\ \frac{}{H\ ;\ \mathsf{skip}\ \Downarrow\ H} \qquad \text{ASSIGN}\ \frac{H\ ;\ e\ \Downarrow\ i}{H\ ;\ x := e\ \Downarrow\ H, x \mapsto i} \qquad \text{SEQ}\ \frac{H_1\ ;\ s_1\ \Downarrow\ H_3 \qquad H_3\ ;\ s_2\ \Downarrow\ H_2}{H_1\ ;\ s_1; s_2\ \Downarrow\ H_2}$$

$$\text{IF1}\ \frac{H_1\ ;\ e\ \Downarrow\ i \quad i \neq 0 \quad H_1\ ;\ s_1\ \Downarrow\ H_2}{H_1\ ;\ \mathsf{if}\ e\ s_1\ s_2\ \Downarrow\ H_2} \qquad \text{IF2}\ \frac{H_1\ ;\ e\ \Downarrow\ 0 \quad H_1\ ;\ s_2\ \Downarrow\ H_2}{H_1\ ;\ \mathsf{if}\ e\ s_1\ s_2\ \Downarrow\ H_2} \qquad \text{WHILE}\ \frac{H_1\ ;\ \mathsf{if}\ e\ (s; \mathsf{while}\ e\ s)\ \mathsf{skip}\ \Downarrow\ H_2}{H_1\ ;\ \mathsf{while}\ e\ s\ \Downarrow\ H_2}$$

$$\begin{array}{rcl} e & ::= & \lambda x.\ e \mid x \mid e\ e \mid c \mid (e, e) \mid e.1 \mid e.2 \mid \{l_1 = e_1, \ldots, l_n = e_n\} \mid e.l_i \\ & \mid & \mathsf{letrec}\ f\ x.\ e \mid \mathsf{A}(e) \mid \mathsf{B}(e) \mid (\mathsf{match}\ e\ \mathsf{with}\ \mathsf{A}x.\ e \mid \mathsf{B}x.\ e) \\ v & ::= & \lambda x.\ e \mid \mathsf{letrec}\ f\ x.\ e \mid c \mid (v, v) \mid \{l_1 = v_1, \ldots, l_n = v_n\} \mid \mathsf{A}(v) \mid \mathsf{B}(v) \\ \tau & ::= & \mathsf{int} \mid \tau \to \tau \mid \tau * \tau \mid \{l_1 : \tau_1, \ldots, l_n : \tau_n\} \mid \tau + \tau \end{array}$$

$\boxed{e_1 \to e_2}$

$$\frac{}{(\lambda x.\ e)\ v \to e\{v/x\}} \qquad \frac{e_1 \to e_1'}{e_1\ e_2 \to e_1'\ e_2} \qquad \frac{e_2 \to e_2'}{v\ e_2 \to v\ e_2'} \qquad \frac{e_1 \to e_1'}{(e_1, e_2) \to (e_1', e_2)} \qquad \frac{e_2 \to e_2'}{(v, e_2) \to (v, e_2')}$$

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

$$\frac{}{(\mathsf{letrec}\ f\ x.\ e)\ v \to (e\{v/x\})\{\mathsf{letrec}\ f\ x.\ e/f\}} \qquad \frac{}{\{l_1 = v_1, \ldots, l_n = v_n\}.l_i \to v_i}$$

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

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

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

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

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

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

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

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

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

$\boxed{\tau_1 \leq \tau_2}$

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

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

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

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

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. (Caml Programming and Type Inference) (**17** points)

   (a) Write a function `wgo_help1` such that this function:

   ```
   let wgo1 max lst = wgo_help1 0 max lst
   ```

   behaves as follows: If `lst` is a list of integers `[i1;i2;...;in]`, then `wgo1` returns the sum of a prefix of the list `i1 ... ij` such that:

   - The sum is less than `max`.
   - Either the sum of the next-larger prefix `i1 ... ij i` is not less than `max` or there is no next-larger prefix (i.e., the entire list has a sum less than `max`).

   Do not define any other helper functions. Note wgo stands for, "without going over."

   (b) What is the type of `wgo_help1`?

   (c) Write a function `wgo_help2` such that this function:

   ```
   let wgo2 max lst = wgo_help2 (fun x -> x < max) (fun x y -> x+y) 0 lst
   ```

   has the same observable behavior as `wgo1`. Do not define any other functions. `wgo_help2` should not contain an explicit addition.

   (d) What is the type of `wgo_help2`?

   (e) Rewrite `wgo1` and `wgo2` to be shorter.

Solution:

(a)
```
let rec wgo_help1 acc max lst =
   match lst with
     [] -> acc
   | hd::tl -> if acc+hd < max
               then wgo_help1 (acc+hd) max tl
               else acc
```

(b) `int -> int -> int list -> int`

(c)
```
let rec wgo_help2 f g acc lst =
   match lst with
     [] -> acc
   | hd::tl -> if f (g hd acc)
               then wgo_help2 f g (g hd acc) tl
               else acc
```

(d) `('a -> bool) -> ('b -> 'a -> 'a) -> 'a -> 'b list -> 'a` Note that it also works to use `(g acc hd)`, which changes the type of the second argument to `'a -> 'b -> 'a`. Substantial partial credit was given for a type with only one type variable.

(e)
```
let wgo1 = wgo_help1 0
let wgo2 max = wgo_help2 (fun x -> x < max) (fun x y -> x+y) 0
```

Name:____

2. (IMP and translation) (**15** points)

   Here is our Caml abstract syntax for IMP with one new kind of statement described below:

```
type exp = Int of int | Var of string | Plus of exp * exp | Times of exp * exp
type stmt = Skip | Assign of string * exp | Seq of stmt * stmt
          | If of exp * stmt * stmt | While of exp * stmt
          | CompareAndSwap of string * exp * exp
```

   Recall that in the semantics the expression in an if-statement or while-statement is true if it is not zero.

   In this problem, we consider a new kind of statement in IMP. The semantics of `CompareAndSwap(s,e1,e2)` is as follows:

   - If evaluating `e1` under the current heap produces the same value that variable `s` holds under the current heap, then update the heap so `s` holds the value that `e2` evaluates to under the current heap.
   - Otherwise make no change to the heap.

   In Caml, write a *translation* from IMP-including-compare-and-swap statements to IMP-not-including-compare-and-swap statements. In other words, write a function `translate` of type `stmt -> stmt` such that (1) the result contains no compare-and-swap statements and (2) the result is equivalent to the argument.

   Note: Some of you might recognize compare-and-swap as related to concurrency, but this problem has nothing to do with concurrency.

   Solution:

```
let rec translate s =
  match s with
    Skip -> s
  | Assign(str,e) -> s
  | Seq(s1,s2) -> Seq(translate s1, translate s2)
  | If(e,s1,s2) -> If(e, translate s1, translate s2)
  | While(e,s) -> While(e, translate s)
  | CompareAndSwap(str,e1,e2) -> If(Plus(e1,Times(Int(-1),Var(str))),
                                    Skip,
                                    Assign(str,e2))
```

Name:____

3. (Formal operational semantics) (**18** points)

In this problem we define a small language that manipulates a stack of strings. You are given the syntax and the informal semantics.

The language syntax is a command list. A program state contains a command list and a stack of strings (the stack "grows to the right"):

$$\begin{array}{rrcl} \text{string} & str & ::= & \textit{(any string)} \\ \text{command} & c & ::= & \mathsf{push}\ str \mid \mathsf{pop} \mid \mathsf{dup} \mid \mathsf{swap} \\ \text{command-list} & lst & ::= & [] \mid c{::}lst \\ \text{stack} & stk & ::= & \cdot \mid stk, str \end{array}$$

Informally, the commands behave as follows:

- push *str* makes a bigger stack with *str* on top.
- pop makes a smaller stack by removing the top element.
- dup (short for duplicate) makes a bigger stack by placing a copy of the top stack-element on top.
- swap swaps the order of the top two elements on the stack.

A command list executes the commands in order.

(a) Give large-step inference rules for the judgment $stk_1; lst \Downarrow stk_2$, meaning, "running *lst* starting from stk_1 produces stk_2." One rule is given to you as an example. You need to write down 4 other rules.

$$\frac{stk_1, str;\ lst \Downarrow stk_2}{stk_1;\ (\mathsf{push}\ str){::}lst \Downarrow stk_2}$$

(b) The semantics can *get stuck*, i.e., there exists stacks stk_1 and command-lists *lst* such that we cannot derive $stk_1; lst \Downarrow stk_2$ for any stk_2. In English, describe why there may be not be a derivation.

(c) Give a complete derivation that concludes $\cdot; (\mathsf{push}\ \text{"pl"}){::}\mathsf{dup}{::}\mathsf{swap}{::}[] \Downarrow \cdot, \text{"pl"}, \text{"pl"}$

Solution:

(a)

$$\frac{stk_1; lst \Downarrow stk_2}{stk_1, str;\ \mathsf{pop}{::}lst \Downarrow stk_2} \qquad \frac{stk_1, str, str; lst \Downarrow stk_2}{stk_1, str;\ \mathsf{dup}{::}lst \Downarrow stk_2} \qquad \frac{stk_1, str_2, str_1; lst \Downarrow stk_2}{stk_1, str_1, str_2;\ \mathsf{swap}{::}lst \Downarrow stk_2}$$

$$\frac{}{stk;\ [] \Downarrow stk}$$

(b) Evaluation could require popping or duplicating when the stack is empty or swapping when the stack has zero or one elements.

(c)

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\cdot, \text{"pl"}, \text{"pl"};\ [] \Downarrow \cdot, \text{"pl"}, \text{"pl"}}}{\cdot, \text{"pl"}, \text{"pl"};\ \mathsf{swap}{::}[] \Downarrow \cdot, \text{"pl"}, \text{"pl"}}}{\cdot, \text{"pl"};\ \mathsf{dup}{::}\mathsf{swap}{::}[] \Downarrow \cdot, \text{"pl"}, \text{"pl"}}}{\cdot; (\mathsf{push}\ \text{"pl"}){::}\mathsf{dup}{::}\mathsf{swap}{::}[] \Downarrow \cdot, \text{"pl"}, \text{"pl"}}$$

Name:____

This page is intentionally blank since you might (or might not) want more room to answer problem 3.

Name:____

4. (Formal typing rules) (**15** points)

   (a) Recall this typing rule, one of the three rules we added for sums:

   $$\frac{\Gamma \vdash e : \tau_1}{\Gamma \vdash \mathsf{A}(e) : \tau_1 + \tau_2}$$

   Explain this rule in English. In particular, what expressions can this typing rule be used for and what types can it give to such expressions?

   (b) Recall this typing rule for functions:

   $$\frac{\Gamma, x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash \lambda x.\ e : \tau_1 \rightarrow \tau_2}$$

   Explain this rule in English. In particular, what expressions can this typing rule be used for and what types can it give to such expressions?

   (c) Suppose we changed the typing rule for functions to the following:

   $$\frac{\Gamma, x : \tau_2 \vdash e : \tau_2}{\Gamma \vdash \lambda x.\ e : \tau_2 \rightarrow \tau_2}$$

   Explain why this change would not violate type safety. Explain why it is a bad idea anyway.

Solution:

(a) This typing rule applies to any expression that uses "tag" A with a subexpression that type-checks under some context. The overall expression then type-checks under the same context. If the subexpression has type τ_1, then the overall expression has type $\tau_1 + \tau_2$ for any type τ_2.

(b) The typing rule applies to any lambda expression, provided the body type-checks when extending the context to map x to some type τ_1. If the body can have type τ_2, then the function can have type $\tau_1 \rightarrow \tau_2$.

(c) This change would require the argument and result type for every function to be the same type. As examples, we could have $\mathsf{int} \rightarrow \mathsf{int}$ and $(\mathsf{int} * \mathsf{int}) \rightarrow (\mathsf{int} * \mathsf{int})$, but not $(\mathsf{int} * \mathsf{int}) \rightarrow \mathsf{int}$. This is type-safe: Before the change, only safe programs were accepted and the change causes the type-system to accept only strictly fewer programs. Nonetheless, it is a bad idea because functions that take and return value of different types are useful and we would not want to program in a language that did not allow such functions.

Name:____

5. (Soundness/Completeness) (**10** 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 there are exactly 47 programs that your type system accepts; it rejects all others.

   Explain your answers *briefly*.

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

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

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

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

   Solution:

   (a) Yes, soundness means the type system accepts no programs that dereference null. That might be the case for the 47 accepted programs.

   (b) No, completeness means the type system rejects no programs that definitely do not dereference null. There are an infinite number of such programs, and we are rejecting all but 47 of them.

   (c) No, some of the 47 programs might be able to dereference null.

   (d) No, it is not even possible (see part (b)).

Name:____

6. (Concurrent ML) (**13** points)

In this problem you will use Concurrent ML to write a "server version" of `wgo` from problem 1. You will implement this interface:

```
type wgo_adder
val new_wgo_adder : int -> wgo_adder
val add : wgo_adder -> int -> int
```

The argument to `new_wgo_adder` is the new adder's "max." Initially an adder has "not reached its max." While an adder has "not reached its max" each call to `add` returns the sum of all integers ever passed to `add` with this particular adder. However, if the argument to `add` plus the sum of all previous integers exceeds "max" then the "max is reached" and `add` should return the sum of all *previous* calls. After the "max is reached" all future calls to `add` with this adder return this same sum no matter what.

You must use Concurrent ML to support multiple threads calling `add`; do not use other synchronization mechanisms such as locks.

Hints: You do not need `choose` and `wrap`. The code you wrote in problem 1 is *not* particularly helpful.

Solution:

```
type wgo_adder = int channel * int channel

let new_wgo_adder max =
   let inc = new_channel() in
   let outc = new_channel() in
   let rec is_done sum =
       let _ = sync (receive inc) in
       sync (send outc sum);
       is_done sum in
   let rec not_done sum =
       let i = sync (receive inc) in
       if sum + i < max
       then (sync (send outc (sum + i)); not_done (sum + i))
       else (sync (send outc sum); is_done sum) in
   ignore(Thread.create not_done 0); (inc,outc)

let add (c1,c2) i =
   sync (send c1 i);
   sync (receive c2)
```

Note this problem can be solved without two helper functions, but you have to be careful since integers can be negative. A solution that works for nonnegative numbers gets substantial credit.

Name:____

7. (OOP overloading and casting) (**12** points)

   Consider a typical class-based OOP language like we did in class. Suppose somewhere in a program that type-checks we have `e.m((D)(new C()))` where `C` is a subclass/subtype of `D`. Notice the argument in the method call is an explicit upcast. Consider modifying the program by removing this explicit upcast, i.e., replacing the call with `e.m(new C())`.

   Explain your answers *briefly*.

   (a) If every class in the program has at most one method named `m`, can this change cause the program not to type-check?

   (b) If every class in the program has at most one method named `m`, can this change cause the program to produce a different result?

   (c) If classes can have multiple methods named `m` and method calls are resolved with static overloading, can this change cause the program to produce a different result?

   (d) If classes can have multiple methods named `m` and method calls are resolved with multimethods, can this change cause the program to produce a different result?

   Solution:

   (a) No. The program type-checked when the argument had type `D` and it can still have type `D` via subsumption since `C`$\leq$`D`.

   (b) No, the same method would still be called with the same object.

   (c) Yes, if the class of the object `e` evaluates to has a method named `m` that takes a `C` and another that takes a `D`, then the change will change which method is executed for this call. This is because the type of the argument is different after removing the explicit upcast.

   (d) No, multimethods resolve method calls using the run-time class of arguments. The explicit upcast has no run-time effect, so the data used to resolve the method call is the same.

   Note in all cases we pass an instance of `C` to whatever method we are calling, so the cast cannot affect how the body of the called method executes. It can, as the answers above describe, affect which method is called.