CSE P505, Spring 2006, Final Examination
6 June 2006

Rules:

• Please do not turn the page until everyone is ready.
• The exam is closed-book, closed-note, except for two sides of one 8.5x11in piece of paper.
• Please stop promptly at 8:30.
• You can rip apart the pages, but please write your name on each page.
• There are 100 points total, distributed very 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 and focus on the questions worth more points.
• If you have questions, ask.
• Relax. You are here to learn.
For your reference (page 1 of 2):

\[
\begin{array}{llll}
\text{CONST} & \text{VAR} & \text{ADD} & \text{MULT} \\
H; e \downarrow c & H; x \downarrow H(x) & H; e_1 \downarrow c_1 \quad H; e_2 \downarrow c_2 & H; e_1 + e_2 \downarrow c_1 + c_2 \\
\end{array}
\]

\[
\begin{array}{llll}
\text{SKIP} & \text{ASSIGN} & \text{SEQ} & \text{WHILE} \\
H; \text{skip} \downarrow H & H; e \downarrow i & H; s_1 \downarrow H'' & H; \text{if } e \text{ skip } \downarrow H' \\
H; \text{if } e s_1 s_2 \downarrow H' & H; \text{if } e s_1 s_2 \downarrow H' & H; \text{if } e s_1 s_2 \downarrow H' & H; \text{while } e s \downarrow H' \\
\end{array}
\]

\[
\begin{array}{llll}
\lambda x. e \downarrow \lambda x. e & e_1 \downarrow \lambda x. e_3 & e_2 \downarrow v_2 & e_3 \{v_2/x\} \downarrow v \\
& e_1 e_2 \downarrow v & & \end{array}
\]

\[
\begin{array}{ll}
FV(x) &= \{x\} \\
FV(e_1 e_2) &= FV(e_1) \cup FV(e_2) \\
FV(\lambda x. e) &= FV(e) - \{x\} \\
\end{array}
\]

\[
\begin{array}{llll}
e \downarrow v & e \downarrow v & e \downarrow e' & e \downarrow e' \\
v e \downarrow v e' & e[\tau] \downarrow e'[\tau] & (\lambda x: \tau. e) v \downarrow e[v/x] & (\Lambda \alpha. e)[\tau] \rightarrow e[\tau/\alpha] \\
\end{array}
\]

\[
\begin{array}{ll}
\Gamma \vdash x : \Gamma(x) & \Gamma \vdash c : \text{int} \\
\end{array}
\]

\[
\begin{array}{ll}
\Gamma, x: \tau_1 \vdash e : \tau_2 & \Gamma \vdash \tau_1 \\
\Gamma \vdash \lambda x: \tau_1. e : \tau_1 \rightarrow \tau_2 & \Gamma \vdash \Lambda \alpha. e : \forall \alpha. \tau_1 \\
\Gamma \vdash e_1 : \tau_2 \rightarrow \tau_1 & \Gamma \vdash e_2 : \tau_2 \\
\Gamma \vdash e_1 : \tau_1 & \Gamma \vdash e_2 : \tau_1 \\
\Gamma \vdash e : \forall \alpha. \tau_1 & \Gamma \vdash e_\tau_2 : \tau_1{\tau_2/\alpha} \\
\end{array}
\]
Name:

\[
e := \lambda x. e \mid e \mid e \mid c \mid (e, e) \mid e.1 \mid e.2 \mid A \mid B \mid c | \text{match } e \text{ with } A \to B | x \to e | \text{letrec } f. x. e
\]

\[
v := \lambda x. e \mid c \mid (v, v) \mid A \mid B \mid v \mid \{l_1 = v, \ldots, l_n = v\}
\]

\[
\tau := \text{int} \mid \tau \to \tau \mid \tau * \tau \mid \tau + \tau \mid \{l_1 = \tau, \ldots, l_n = \tau\}
\]

\[
e \to e'
\]

\[
\begin{array}{c}
e_1 \to e'_1 \\
\frac{e_1 e_2 \to e'_1 e_2}{v \to e' \cdot v e' \cdot (\lambda x. e) \to e\cdot v / x} \\
\frac{v \to e' \cdot e_1 \to e'_1}{(e_1, e_2) \to (e'_1, e_2)} \\
\frac{v \to \cdot v e' \cdot e_2 \to e'_2}{e_2 \to e'_2} \\
\frac{e \to e'}{e \to e'} \\
\end{array}
\]

\[
\frac{e \to e'}{A \to A} \\
\]

\[
\frac{B \to B}{e \to e'}
\]

\[
\text{match } e_1 \text{ with } A \to e_2 \mid B \\ \text{y} \to e_3 \to \text{match } e'_1 \text{ with } A \to e_2 \mid B \\ y \to e_3
\]

\[
\frac{\{l_1 = v_1, \ldots, l_n = v_n\} \cdot l_i \to v_i}{e_i \to e'_i}
\]

\[
\{l_1 = v_1, \ldots, l_i-1 = v_i-1, l_i = e_i, \ldots, l_n = e_n\} \to \{l_1 = v_1, \ldots, l_i-1 = v_i-1, l_i = e'_i, \ldots, l_n = e_n\}
\]

\[
\Gamma \vdash e : \tau \text{ and } \tau_1 \leq \tau_2
\]

\[
\begin{array}{c}
\Gamma \vdash e : \text{int} \\
\frac{\Gamma \vdash x : \Gamma(x)}{\Gamma \vdash \lambda x. e : \tau_1 \rightarrow \tau_2} \\
\frac{\Gamma \vdash \text{letrec } f. x. e : \tau_1 \rightarrow \tau_2}{\Gamma \vdash \text{letrec } f. x. e : \tau_1 \rightarrow \tau_2} \\
\end{array}
\]

\[
\begin{array}{c}
\frac{\{l_1 = \tau_1, \ldots, l_n = \tau_n\}}{\Gamma \vdash \{l_1 = e_1, \ldots, l_n = e_n\} : \{l_1 = \tau_1, \ldots, l_n = \tau_n\}} \\
\frac{\{l_1 = e_1, \ldots, l_n = e_n\} : \{l_1 = \tau_1, \ldots, l_n = \tau_n\} \mid 1 \leq i \leq n}{\Gamma \vdash e : \{l_1 = \tau_1, \ldots, l_n = \tau_n\}} \\
\frac{\{l_1 = e_1, \ldots, l_n = e_n\} : \{l_1 = \tau_1, \ldots, l_n = \tau_n\} \mid 1 \leq i \leq n}{\Gamma \vdash e : \{l_1 = \tau_1, \ldots, l_n = \tau_n\}} \\
\end{array}
\]

\[
\frac{\Gamma \vdash e : \tau \quad \tau \leq \tau'}{\Gamma \vdash e : \tau'} \\
\frac{\tau_1 \leq \tau_2 \quad \tau_2 \leq \tau_3 \quad \tau_3 \leq \tau_1}{\tau_1 \rightarrow \tau_2 \leq \tau_3 \rightarrow \tau_4}
\]

\[
\begin{array}{c}
\{l_1 = \tau_1, \ldots, l_n = \tau_n, l = \tau\} \leq \{l_1 = \tau_1, \ldots, l_n = \tau_n\} \\
\{l_1 = \tau_1, \ldots, l_i = \tau_i, l_j = \tau_j, \ldots, l_n = \tau_n\} \leq \{l_1 = \tau_1, \ldots, l_i = \tau_i, l_j = \tau_j, \ldots, l_n = \tau_n\}
\end{array}
\]

\[
\frac{\tau_1 \leq \tau_1'}{\{l_1 = \tau_1, \ldots, l_i = \tau_i, \ldots, l_n = \tau_n\} \leq \{l_1 = \tau_1, \ldots, l_i = \tau_i', \ldots, l_n = \tau_n\}}
\]
1. (20 points) Suppose we add division to our IMP expression language. In Caml, the expression syntax becomes:

```
type exp =
  Int of int | Var of string | Plus of exp * exp | Times of exp * exp | Div of exp * exp
```

Our interpreter (not shown) raises a Caml exception if the second argument to `Div` evaluates to 0. We are ignoring statements; assume an IMP program is an expression that takes an unknown heap and produces an integer.

(a) Write a Caml function `nsz` (stands for “no syntactic zero”) of type `exp->bool` that returns false if and only if its argument contains a division where the second argument is the integer constant 0. Note we are not interpreting the input; `nsz` is not even passed a heap.

(b) If we consider division-by-zero at run-time a “stuck state” and `nsz` a “type system” (where true means “type-checks”), then:
   i. Is `nsz` sound? Explain.
   ii. Is `nsz` complete? Explain.
2. (20 points) Consider this Caml code. It uses `strcmp`, which has type `string->string->bool` and the expected behavior.

```caml
exception NoValue
let empty = fun s -> raise NoValue
let extend m x v = fun s -> if strcmp s x then v else m s
let lookup m x = m x
```

(a) What functionality do these three bindings provide a client?
(b) What types do each of the bindings have?
   (Note: They are all polymorphic and may have more general types than expected.)
3. (16 points) When we added sums (syntax $A e$, $B e$, and $\text{match } e_1 \text{ with } A x \rightarrow e_2 | B y \rightarrow e_3$) to the $\lambda$-calculus, we gave a small-step semantics and had exactly two constructors.

(a) Give sums a large-step semantics, still for exactly two constructors. That is, extend the call-by-value large-step judgment $e \Downarrow v$ with new rules. (Use 4 rules.)

(b) Suppose a program is written with three constructors ($A$, $B$, and $C$) and match expressions that have exactly three cases:

$$\text{match } e_1 \text{ with } A x \rightarrow e_2 | B y \rightarrow e_3 | C z \rightarrow e_4$$

Explain a possible translation of such a program into an equivalent one that uses only two constructors. (That is, explain how to translate the 3 constructors to use 2 constructors and how to translate match expressions. Do not write inference rules.)
4. (14 points) Consider a \( \lambda \)-calculus with tuples (i.e., “pairs with any number of fields”), so we have expressions \((e_1, e_2, \ldots, e_n)\) and \(e.i\) and types \(\tau_1 \ast \tau_2 \ast \cdots \ast \tau_n\). For each of our subtyping rules for records, explain whether or not an analogous rule for tuples makes sense.
5. (14 points) Assume a class-based object-oriented language as in class, and a program that contains the call \texttt{e.f((C)e1)} where \texttt{e1} is a (compile-time) subtype of \texttt{C} and the whole call type-checks.

(a) If calls are resolved with static overloading, is it possible that removing the cast \texttt{C} (i.e., changing the call to \texttt{e.f(e1)}) could cause the program to still type-check but behave differently? Explain.

(b) If calls are resolved with static overloading and we have multiple inheritance, is it possible that removing the cast \texttt{C} (i.e., changing the call to \texttt{e.f(e1)}) could cause the program to no longer type-check? Explain.

(c) If calls are resolved with multimethods, is it possible that removing the cast \texttt{C} (i.e., changing the call to \texttt{e.f(e1)}) could cause the program to behave differently? Explain.
6. (9 points) Here are two large-step interpreters for the untyped lambda-calculus. The one on the right uses parallelism. Recall \texttt{Thread.join} blocks until the thread described by its argument terminates. Only the lines between the (*----------*) comments differ.

```ocaml
type exp = Var of string | Lam of string*exp | Apply of exp * exp

exception UnboundVar

let subst e1_with e2_for x = ... (* unimportant *)

let rec interp e = let rec interp e =
match e with match e with
| Var _ -> raise UnboundVar | Var x -> raise UnboundVar
| Lam _ -> e | Lam _ -> e
| Apply(e1,e2) -> | Apply(e1,e2) ->
  (*----------*) (*----------*)

let v2r = ref (Var "dummy") in
let t = Thread.create
  (fun () -> v2r := interp e2) ()
in
let v1 = interp e1 in
let Thread.join t;
let v2 = !v2r in

match v1 with match v1 with
| Lam(x,e3) -> interp(subst e3 v2 x) | Lam(x,e3) -> interp(subst e3 v2 x)
| _ -> failwith "impossible" | _ -> failwith "impossible"
```

(a) Describe an input to these functions for which the interpreter on the right would raise an exception and the interpreter on the left would not. (Note: Evaluation of expressions may not terminate.)

(b) Explain why moving the line "let v2r = ref (Var "dummy") in" out to the top-level (and removing the keyword "in") would make the interpreter on the right behave unpredictably (even for inputs with no free variables).
7. **(7 points)** You can do this problem in one of Caml, C, C++, Java, or C#. Your choice does not really change the problem.

(a) Write a short program that will exhaust memory if there is no garbage collector but take almost no space if there is a garbage-collector.

(b) Write a short program that will exhaust memory even if there is a garbage collector. Create only small objects.