Name:____

CSE 505, Fall 2003, Midquarter Examination
4 November 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 11:50.
- 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.

Name:____

1. Consider this syntax for IMP expressions, which *has (integer) division as the only arithmetic operator*:

$$e ::= c \mid x \mid e/e$$

   (a) Give a large-step operational semantics of the form $H\ ;\ e \Downarrow c$ for these expressions. Make sure that if evaluation of e under H would involve dividing by 0, then there is no c for which you can derive $H\ ;\ e \Downarrow c$.
   (Hint: You need 3 inference rules.)
   (Note: You may assume $H(x)$ is defined as in class.)

   (b) Now suppose we add an explicit **error** result. Add inference rules to your previous answer so that $H\ ;\ e \Downarrow v$ where $v ::= c \mid \textbf{error}$. Make sure that if evaluation of e under H would involve dividing by 0, then $H\ ;\ e \Downarrow \textbf{error}$.
   (Hint: You need 3 more inference rules, so 6 total.)

   (c) Does adding the rule $\dfrac{}{H\ ;\ 0/e \Downarrow 0}$ change the semantics you defined for (a) and (b)? Explain.

Solution:

(a)

$$\frac{}{H\ ;\ c \Downarrow c} \qquad \frac{}{H\ ;\ x \Downarrow H(x)} \qquad \frac{H\ ;\ e_1 \Downarrow c_1 \quad H\ ;\ e_2 \Downarrow c_2 \quad c_2 \neq 0}{H\ ;\ e_1/e_2 \Downarrow c_1/c_2}$$

(The conclusions of the last rule uses the "math" /. I didn't count off for omitting $c_2 \neq 0$ because you could claim the math / simply does not apply if c_2 is 0.)

(b) We add:

$$\frac{H\ ;\ e_1 \Downarrow \textbf{error}}{H\ ;\ e_1/e_2 \Downarrow \textbf{error}} \qquad \frac{H\ ;\ e_2 \Downarrow \textbf{error}}{H\ ;\ e_1/e_2 \Downarrow \textbf{error}} \qquad \frac{H\ ;\ e_2 \Downarrow 0}{H\ ;\ e_1/e_2 \Downarrow \textbf{error}}$$

(c) Yes, this rule would let us derive results like $H\ ;\ 0/0 \Downarrow 0$, which the solutions to parts (a) and (b) do not allow.

Name:____

2. Here is our *unchanged* syntax and semantics for IMP statements:

$$s ::= \mathsf{skip} \mid x := e \mid s; s \mid \mathsf{if}\ e\ s\ s \mid \mathsf{while}\ e\ s$$

$$\textsc{assign}\ \frac{H\ ;\ e \Downarrow c}{H\ ;\ x := e \to H, x \mapsto c\ ;\ \mathsf{skip}} \qquad \textsc{seq1}\ \frac{}{H\ ;\ \mathsf{skip}; s \to H\ ;\ s} \qquad \textsc{seq2}\ \frac{H\ ;\ s_1 \to H'\ ;\ s_1'}{H\ ;\ s_1; s_2 \to H'\ ;\ s_1'; s_2}$$

$$\textsc{if1}\ \frac{H\ ;\ e \Downarrow c \qquad c>0}{H\ ;\ \mathsf{if}\ e\ s_1\ s_2 \to H\ ;\ s_1} \qquad \textsc{if2}\ \frac{H\ ;\ e \Downarrow c \qquad c\leq 0}{H\ ;\ \mathsf{if}\ e\ s_1\ s_2 \to H\ ;\ s_2} \qquad \textsc{while}\ \frac{}{H\ ;\ \mathsf{while}\ e\ s \to H\ ;\ \mathsf{if}\ e\ (s; \mathsf{while}\ e\ s)\ \mathsf{skip}}$$

(a) Define a judgment of the form $\mathrm{mysize(s) = n}$. Informally, n should be: (the number of skip statements in s) plus (*two times* the number of assignment statements in s). For example, you should be able to derive $\mathrm{mysize}(\mathsf{skip}; \mathrm{x := 0; y := 1}) = 5$.
(Hint: You need 5 inference rules.)

(b) Prove the following: If s *has no while-statements or if-statements* and $H\ ;\ s \to H'\ ;\ s'$ and $\mathrm{mysize(s) = n}$ and $\mathrm{mysize(s') = n'}$, then $n' < n$.
Note: This theorem is true with if-statements (but not while-statements), but you do not have to show this.

(c) Using part (b), argue *informally* (no proof required) that while-free programs terminate.

Solution:

(a)

$$\frac{}{\mathrm{mysize}(\mathsf{skip}) = 1} \qquad \frac{}{\mathrm{mysize(x := e) = 2}} \qquad \frac{\mathrm{mysize(s_1) = n_1} \qquad \mathrm{mysize(s_2) = n_2}}{\mathrm{mysize(s_1; s_2) = n_1 + n_2}}$$

$$\frac{\mathrm{mysize(s_1) = n_1} \qquad \mathrm{mysize(s_2) = n_2}}{\mathrm{mysize}(\mathsf{if}\ \mathrm{e\ s_1\ s_2}) = \mathrm{n_1 + n_2}} \qquad \frac{\mathrm{mysize(s) = n}}{\mathrm{mysize}(\mathsf{while}\ \mathrm{e\ s}) = \mathrm{n}}$$

(b) The proof is by induction on the derivation of $H\ ;\ s \to H'\ ;\ s'$, proceeding by cases on the bottom-most rule:

- If s is some $x := e$ then n is 2 and n' is 1.
- If s is some $s_1; s_2$ and s_1 is not skip, then we have a shorter derivation of $H\ ;\ s_1 \to H'\ ;\ s_1'$. Furthermore, $\mathrm{mysize(s_1; s_2) = n_1 + n_2}$ where $\mathrm{mysize(s_1) = n_1}$ and $\mathrm{mysize(s_2) = n_2}$. So by induction $\mathrm{mysize(s_1') = n_1'}$ where $n_1' < n_1$. So we can derive $\mathrm{mysize(s_1'; s_2) = n_1' + n_2}$ and $n_1' + n_2 < n_1 + n_2$.
Note: We're implicitly using a lemma that $\mathrm{mysize(s) = n}$ implies $n > 0$, but I didn't take off if you failed to say that explicitly.
- If s has the form $\mathsf{skip}; s_2$, then s' is s_2 and inverting $\mathrm{mysize(s) = n}$ ensures $\mathrm{mysize(s_2) = n - 1}$. Clearly $n - 1 < n$.
- If s if an if-statement or while-loop, the lemma holds vacuously.

(c) If s is while-free and $\mathrm{mysize(s) = n}$, then the previous part proved the size of s decreases on every step. So in at most n steps its size must be 1, which means it has become skip.

Name:____

3. Describe what each of the following O'Caml programs would print:

   (a)
   ```
   let f x y = x y in
   let z = f print_string "hi" in
   f print_string "hi"
   ```

   (b)
   ```
   let f x = (fun y -> print_string x) in
   let g = f "hi" in
   let x = "mom" in
   g "pizza"
   ```

   (c)
   ```
   let rec f n x =
     if n>0
     then (let _ = print_string x in f (n-1) x)
     else ()
   in
   f 3 "hi"
   ```

   (d)
   ```
   let rec f n x =
     if n>0
     then (let _ = print_string x in f (n-1) x)
     else ()
   in
   f 3
   ```

   (e)
   ```
   let rec f x = f x in
   print_string (f "hi")
   ```

Solution:

(a) "hi" "hi"

(b) "hi"

(c) "hi" "hi" "hi"

(d) prints nothing (evaluates to a function that prints when called)

(e) prints nothing (goes into an infinite loop)

Name:____

4. Consider a λ-calculus with *pairs* built-in. That is, (v_1, v_2) is a value if v_1 and v_2 are values, $(v_1, v_2).1 \to v_1$ and $(v_1, v_2).2 \to v_2$.

   (a) Give an encoding of triples that uses pairs. You should define four terms: a three-argument function (using currying) to build a triple, and functions for returning the first, second, and third part of a triple. (By *encoding*, we mean you may *not* extend the syntax of the language.)

   (b) In the simply-typed λ-calcus with pairs (and types of the form $\tau_1 * \tau_2$), give *two different types* that your function for forming a triple could have. (I.e., if e is your term for building a triple, give two τ such that $\cdot \vdash e : \tau$.)

Solution:

(a) "make-triple" $= \lambda x.\ \lambda y.\ \lambda z.\ ((x, y), z)$
"first" $= \lambda x.\ x.1.1$
"second" $= \lambda x.\ x.1.2$
"third" $= \lambda x.\ x.2$

(b) `int->int->int->((int*int)*int)`
`int->int->(int*int)->((int*int)*(int*int))`
$\ldots$

Name:____

5. Under what assumptions do the following terms type-check in the simply-typed λ-calculus? That is, for the given e, describe all Γ and τ such that $\Gamma \vdash e : \tau$.

   (a) $e = x\ y$

   (b) $e = \lambda x.\ (f\ (f\ x))$

   (c) $e = \lambda x.\ (\lambda y.\ x)$

   (d) $e = \lambda x.\ (x\ (\lambda y.\ x))$

   Solution:

   (a) Any Γ and τ where Γ maps x to a type of the form $\tau_1 \to \tau$ and y to a type of the form τ_1.

   (b) Any Γ and τ where Γ maps f to a type of the form $\tau_1 \to \tau_1$ and $\tau = \tau_1 \to \tau_1$.

   (c) Any Γ and τ where τ has the form $\tau_1 \to \tau_2 \to \tau_1$.

   (d) There is no Γ and τ for which this program type-checks.

Name:____

6. Recall how we extend the simply-typed λ-calculus with *fix*:

$$\frac{e \to e'}{\mathit{fix}\ e \to \mathit{fix}\ e'} \qquad \frac{}{\mathit{fix}\ \lambda x.\ e \to e[(\mathit{fix}\ \lambda x.\ e)/x]} \qquad \frac{\Gamma \vdash e : \tau \to \tau}{\Gamma \vdash \mathit{fix}\ e : \tau}$$

Also we recall that this extension is type-safe.

(a) If we add the rule

$$\frac{}{\Gamma \vdash \mathit{fix}\ e : \tau}$$

is our language still type-safe? If not, give a program that gets stuck. If so, argue the case of the Preservation Lemma proof for a typing derivation ending with this rule.

(b) If we add the rule

$$\frac{}{\Gamma \vdash \mathit{fix}\ \lambda x.\ x : \tau}$$

is our language still type-safe? If not, give a program that gets stuck. If so, argue the case of the Preservation Lemma proof for a typing derivation ending with this rule.

Hint: The Preservation Lemma is: If $\cdot \vdash e : \tau$ and $e \to e'$, then $\cdot \vdash e' : \tau$. We prove it by induction on the derivation of $\cdot \vdash e : \tau$.

Solution:

(a) The extension is not safe because it accepts any term of the form *fix* e. So *fix* (3 4) would get stuck.

(b) The language is safe. For Preservation, if the typing derivation ends with $\frac{}{\cdot \vdash \mathit{fix}\ \lambda x.\ x : \tau}$, we need to show $\cdot \vdash e' : \tau$ if $\mathit{fix}\ \lambda x.\ x \to e'$. But $\mathit{fix}\ \lambda x.\ x \to \mathit{fix}\ \lambda x.\ x$ so the assumed typing derivation is exactly what we need.