CSE 505, Fall 2008, Midterm Examination
29 October 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 1:20.
• You can rip apart the pages, but please write your name on each page.
• There are 100 points total, distributed unevenly among 4 questions (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. 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:

\[ s ::= \text{skip} \mid x := e \mid s \mid \text{if } e \ s \mid \text{while } e \ s \]
\[ e ::= c \mid x \mid e + e \mid e \ast e \]
\[ (c \in \{\ldots, -2, -1, 0, 1, 2, \ldots\}) \]
\[ (x \in \{x_1, x_2, \ldots, y_1, y_2, \ldots, z_1, z_2, \ldots\}) \]

\[ H : e \Downarrow c \]

\begin{align*}
\text{CONST} & \quad \text{VAR} & \quad \text{ADD} & \quad \text{MULT} \\
H ; c \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 \\
H_1 ; s_1 \rightarrow H_2 ; s_2 & & & & H ; e_1 * e_2 \Downarrow c_1 * c_2
\end{align*}

\[ H_1 : s_1 \rightarrow H_2 ; s_2 \]

\[ \text{ASSIGN} \quad \text{SEQ1} \quad \text{SEQ2} \]
\[ H ; e \Downarrow c \quad \text{skip} \quad H ; \text{skip}; s \rightarrow H ; s \]
\[ H ; x := e \rightarrow H, x := c \quad \text{if } e \quad \text{if } s \rightarrow e \rightarrow e' \quad \text{while } \]
\[ H ; e \Downarrow c \quad c > 0 \quad H ; e \Downarrow c \quad c \leq 0 \quad H ; \text{while } e \rightarrow H ; s_2 \]
\[ H ; \text{if } e \ s_1 s_2 \rightarrow H ; s_1 \quad \text{if } e \ s_2 \rightarrow H ; s_2 \quad \text{if } e (s; \text{while } e) \rightarrow \text{skip} \]
\[ e ::= \lambda x. \ e \mid x \ | e \ e \mid c \]
\[ v ::= \lambda x. \ e \mid c \]
\[ \tau ::= \text{int} \mid \tau \rightarrow \tau \]

\[ e \rightarrow e' \]

\[ (\lambda x. \ e) \ v \rightarrow e[v/x] \quad e_1 \rightarrow e'_1 \quad e_2 \rightarrow e'_2 \]
\[ e_1 e_2 \rightarrow e'_1 e'_2 \quad v e_2 \rightarrow v e'_2 \]
\[ e[e'/x] = e'' \]
\[ e_1[e/x] = e_1' \quad y \neq x \quad y \not\in \text{FV}(e) \]
\[ e_1[e/x] = e_1' \quad e_2[e/x] = e_2' \]
\[ (\lambda y. \ e_1)[e/x] = \lambda y. \ e_1' \]
\[ (e_1 e_2)[e/x] = e_1' e_2' \]

\[ \Gamma \vdash e : \tau \]
\[ \Gamma \vdash e : \tau \quad \Gamma, x : \tau_1 \vdash e : \tau_2 \]
\[ \Gamma \vdash e : \tau_2 \rightarrow \tau_1 \quad \Gamma \vdash e_1 e_2 : \tau_2 \]
\[ \Gamma \vdash e : \tau \quad \Gamma \vdash x : \Gamma(x) \quad \Gamma \vdash \lambda x. \ e : \tau_1 
\]
\[ \rightarrow \quad \Gamma \vdash e : \tau \quad \Gamma \vdash e' : \tau' \quad \Gamma \vdash e' : \tau' \quad \Gamma \vdash e[e'/x] : \tau. \]

- Preservation: If \( \vdash e : \tau \) and \( e \rightarrow e' \), then \( \vdash e' : \tau \).
- Progress: If \( \vdash e : \tau \), then \( e \) is a value or there exists an \( e' \) such that \( e \rightarrow e' \).
- Substitution: If \( \Gamma, x : \tau' \vdash e : \tau \) and \( \vdash e' : \tau' \), then \( \Gamma \vdash e[e'/x] : \tau \).
1. This problem (and the next one) involve a tiny language for, “moving around the integer number line.”

Syntax:

```latex
\begin{align*}
  e & ::= \text{go } c \mid \text{reverse } \mid e; e \\
  d & ::= R \mid L \\
  (c & \in \{\ldots,-2,-1,0,1,2,\ldots\})
\end{align*}
```

Informal semantics: A “position” is a direction \(d\) and an integer \(c\). The direction \(R\) is “to the right,” i.e., toward greater numbers, and \(L\) is “to the left,” i.e., toward lesser numbers. An expression “changes the position” as follows:

- \(\text{go } c\) has no effect on the position, if \(c\) is negative.
- \(\text{go } c\) moves the position distance \(c\) in the current direction and does not change the direction, if \(c\) is non-negative. For example, if the “position” is \(R\); 3, then \(\text{go } 2\) produces position \(R; 5\).
- \(\text{reverse}\) changes the direction only.
- \(e_1; e_2\) does \(e_1\) and then \(e_2\).

(a) (12 points) Give a large-step operational semantics for our language. The judgment should have the form \(d; c \downarrow d'; c'\) where \(d\) and \(c\) comprise the “starting position” and \(d'\) and \(c'\) comprise the “ending position.” Do not use any other definitions, functions, or judgments. Hint: Sample solution uses 6 rules and uses “math’s” +, -, <, and ≥.

(b) (14 points) Prove this theorem: If \(e\) has no \(\text{reverse}\) expressions in it and \(R; c; e \downarrow d'; c'\), then \(c' \geq c\). You need a stronger induction hypothesis; be sure to state it clearly.

(c) (4 points) Describe exactly where your proof in part (b) relies on the stronger induction hypothesis.

Solution:

(a)

```
\begin{align*}
  \text{NEG} & \quad \frac{c' < 0}{d; c; \text{go } c' \downarrow d; c} \\
  \text{GOR} & \quad \frac{c \geq 0}{R; c; \text{go } c' \downarrow R; c + c'} \\
  \text{GOL} & \quad \frac{c \geq 0}{L; c; \text{go } c' \downarrow L; c - c'} \\
  \text{REVR} & \quad \frac{c \geq 0}{R; c; \text{reverse } \downarrow L; c}
\end{align*}
```

(b) Stronger induction hypothesis: If \(e\) has no \(\text{reverse}\) expressions in it and \(R; c; e \downarrow d'; c'\), then \(c' \geq c\) and \(d' = R\). Proof is by induction on the derivation of \(R; c; e \downarrow d'; c',\) proceeding by cases on the bottommost rule used in the derivation:

- **NEG** Then \(d' = R\) and \(c' = c\) because this rule does not change the position.
- **GOR** Then \(d' = R\) (clearly in the rule) and \(c' \geq c\) because \(c'\) is \(c\) plus a nonnegative number.
- **GOL** This rule cannot be used because the initial direction is \(R\).
- **REVR** This rule cannot be used because \(e\) has no \(\text{reverse}\) expressions, so \(e\) cannot be \(\text{reverse}\).
- **REVL** This rule cannot be used because \(e\) has no \(\text{reverse}\) expressions, so \(e\) cannot be \(\text{reverse}\). It also cannot be used because the initial direction is \(R\).

(c) In the **SEQ** case, without the stronger induction hypothesis we do not know \(d_1\) is \(R\), but we need this to apply induction to \(d_1; c_1; e_2 \downarrow d'; c'\).
Name: ________________________________

(This page intentionally blank)
2. In this problem, you will write a Caml interpreter for the language in the previous problem.

(a) (3 points) Give two Caml type definitions to define expressions \( e \) and directions \( d \). (For constants, use the Caml int type.)

(b) (12 points) Write a function \( \text{interp} \) that takes two curried arguments: (1) a pair that is the “starting position” and (2) an expression. Return the pair that is the “ending position.”

(c) (2 points) Give the Caml type of \( \text{interp} \).

(d) (2 points) Implement \( \text{interp}_\text{from} \_\text{origin} \), which takes just an expression and returns the result of running it from an initial position that is at the origin of the number line and facing right. Use partial application.

(e) (1 points) Give the Caml type of \( \text{interp}_\text{from} \_\text{origin} \).

Solution:

(a) \[
\text{type exp = Go of int | Reverse | Seq of exp * exp}
\]
\[
\text{type dir = R | L}
\]

(b) (* Note a solution with nested match expressions is also fine *)
\[
\text{let rec interp (d,c) e =}
\]
\[
\text{match d,e with}
\]
\[
\quad \text{R, Go c' -> if c' < 0 then (d,c) else (R,c+c')}
\]
\[
\quad \text{L, Go c' -> if c' < 0 then (d,c) else (L,c-c')}
\]
\[
\quad \text{R, Reverse -> (L,c)}
\]
\[
\quad \text{L, Reverse -> (R,c)}
\]
\[
\quad _, Seq(e1,e2) -> interp (interp (d,c) e1) e2
\]

(c) \[\text{(dir*int)} \rightarrow \text{exp} \rightarrow \text{(dir*int)}\]

(d) \[\text{let interp}_\text{from} \_\text{origin} = \text{interp} \ (\text{R},0)\]

(e) \[\text{exp} \rightarrow \text{(dir*int)}\]
3. This problem extends IMP statements with this strange new syntax and small-step evaluation rules:

\[
s ::= \ldots \mid s \# s
\]

\[
\frac{H; \text{skip} \# s \rightarrow H; s}{H; s_1 \rightarrow H'; s'_1}
\]

\[
\frac{H; s_1 \# s_2 \rightarrow H'; s_2 \# s'_1}{H; s_1; s_2 \rightarrow H'; s'_1; s'_2}
\]

(a) (6 points) Explain in 1–3 informal but precise English sentences the meaning of \(s_1 \# s_2\).

(b) (3 points) Is IMP still deterministic? Explain briefly.

(c) (3 points) Give an \(H\), \(s_1\), and \(s_2\) such that \(H; s_1\) terminates, \(H; s_2\) terminates, \(H; s_1; s_2\) terminates, but \(H; s_1 \# s_2\) does not terminate.

(d) (3 points) Give an \(H\), \(s_1\), and \(s_2\) such that \(H; s_1\) terminates, \(H; s_2\) terminates, \(H; s_1 \# s_2\) terminates, but \(H; s_1; s_2\) does not terminate.

Solution:

(a) \(s_1 \# s_2\) executes \(s_1\) and \(s_2\) by alternating which substatement takes the next step, starting with \(s_1\). In other words, it interleaves their execution with “time slices” of one execution step. After one statement reaches skip, the other statement finishes executing.

(b) Yes, in fact it is still the case that for all \(H\) and \(s\), either \(s\) is skip or there is exactly one derivation of an execution step.

(c) One answer: Let \(H\) be \(\cdot\), let \(s_1\) be \(x := 1; \text{while } y \text{ skip}\), and let \(s_2\) be \(y := 1\).

(d) One answer: Let \(H\) be \(\cdot\), let \(s_1\) be \(y := 0; y := 0; y := 1\), and let \(s_2\) be \(\text{while } y \text{ skip}\). (Interestingly, changing \(s_1\) to \(y := 0; y := 1\) is still correct, but changing \(s_1\) to \(\text{skip}; y := 1\) is incorrect, even though \(H(y) = 0\).
4. In this problem we add options (like Caml’s None and Some) to the simply-typed λ-calculus, using a “get” primitive instead of pattern-matching.

Syntax:

\[
\begin{align*}
  e & ::= \ldots | \text{None} | \text{Some } e | \text{get } e \\
  v & ::= \ldots | \text{None} | \text{Some } v \\
  \tau & ::= \ldots | \tau \text{ option}
\end{align*}
\]

Operational Semantics:

\[
\begin{array}{c|c|c|c}
\text{SOME} & \text{GET-E} & \text{GET-SOME} & \text{GET-NONE} \\
\hline
\text{Some } e \rightarrow \text{Some } e' & \text{get } e \rightarrow \text{get } e' & \text{get } (\text{Some } v) \rightarrow v & \text{get None } \rightarrow \text{get None}
\end{array}
\]

(a) (4 points) One of the evaluation rules is strange and probably not what you would implement in an actual language. Which rule? What does the rule mean?

(b) (12 points) Give 3 appropriate new typing rules, one for each new form of expression. Your rules should be sound without being unnecessarily restrictive.

(c) (4 points) State the new case of the Canonical Forms Lemma for values of option types. (You do not need to prove this easy lemma.)

(d) (15 points) Extend the proof of the Progress Lemma to account for our additions. Include only the new cases. (Note you are only proving Progress, not Preservation or Substitution though those should also hold.) Hints:

- You need to use the new case of Canonical Forms. Be clear about where you do so.
- The strange evaluation rule will also be important. Be clear about where this is.

Solution:

(a) Rule GET-NONE is strange. It causes any program that tries to evaluate get None not to terminate. A real language would do something like throw an exception.

(b)

\[
\begin{align*}
\Gamma & \vdash \text{None } : \tau \text{ option} \\
\Gamma & \vdash e : \tau \\
\Gamma & \vdash \text{Some } e : \tau \text{ option} \\
\Gamma & \vdash \text{get } e : \tau
\end{align*}
\]

(c) If \( \vdash v : \tau \text{ option} \), then \( v \) is None or there exists a \( v' \) such that \( v \) is Some \( v' \).

(d) Recall the proof is by induction on the structure (syntax height) of \( e \). New cases:

- If \( e \) is None, then \( e \) is a value.
- If \( e \) is Some \( e' \) for some \( e' \), then inverting the derivation of \( \vdash e : \tau \) ensures \( \vdash e' : \tau' \) for some \( \tau' \). So by induction either \( e' \) can take a step or it is a value. If \( e' \) can take a step, then \( e \) can take a step with SOME. If \( e' \) is a value, then \( e \) is a value.
- If \( e \) is get \( e' \) for some \( e' \), then inverting the derivation of \( \vdash e : \tau \) ensures \( \vdash e' : \tau \text{ option} \). So by induction either \( e' \) can take a step or it is a value. If \( e' \) can take a step, then \( e \) can take a step with GET-E. If \( e' \) is a value, then Canonical Forms ensures it is None or Some \( v \) for some \( v \). If it is None, we can take a step with GET-NONE (which is why we have our strange rule; in practice we would probably be stuck). If it is Some \( v \), we can take a step with GET-SOME.
Name:______________________________________

(This page intentionally blank)