CIS 624, Fall 2014, Midterm Examination
12 November 2014

Please do not turn the page until everyone is ready.

Rules:

• The exam is closed-book, limited notes as explained in class and on Piazza.

• Please stop promptly at 17:20.

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

• There are **100 points** total, distributed **unevenly** among **5** questions (which have multiple parts). Optional (extra) credit is clearly marked and is in addition to the 100 points for required questions.

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.

<table>
<thead>
<tr>
<th>Problem</th>
<th>Max points</th>
<th>Points</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>20</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>15</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>36</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>14</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>15</td>
<td></td>
</tr>
<tr>
<td>6 optional</td>
<td>+6</td>
<td></td>
</tr>
<tr>
<td>Total</td>
<td>100 (+6)</td>
<td></td>
</tr>
</tbody>
</table>
For your reference:

**IMP syntax as presented in class:**

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

**IMP large-step semantics as presented in class:**

\[
\begin{align*}
  H & ; e \Downarrow c \\
  \begin{array}{rll}
  \text{CONST} & H & ; c \Downarrow c \\
  \text{VAR} & H & ; x \Downarrow H(x) \\
  \text{ADD} & H ; e_1 \Downarrow c_1 & H ; e_2 \Downarrow c_2 \\
  & \Rightarrow H ; e_1 + e_2 \Downarrow c_1 + c_2 \\
  \text{MULT} & H ; e_1 \Downarrow c_1 & H ; e_2 \Downarrow c_2 \\
  & \Rightarrow H ; e_1 * e_2 \Downarrow c_1 * c_2
  \end{array}
\end{align*}
\]

**IMP small-step semantics as presented in class:**

\[
\begin{align*}
  H_1 & ; s_1 \rightarrow H_2 ; s_2 \\
  \text{ASSIGN} & H ; e \Downarrow c \\
  & \Rightarrow H ; x := e \rightarrow H, x \mapsto c ; \text{skip} \\
  \text{SEQ1} & H ; \text{skip} ; s \rightarrow H ; s \\
  \text{SEQ2} & H ; s_1 ; s_2 \rightarrow H' ; s_1' ; s_2 \\
  \text{IF1} & H ; e \Downarrow c \\
  & c > 0 \\
  & \Rightarrow H ; \text{if } e \ s_1 \rightarrow H ; s_1 \\
  \text{IF2} & H ; e \Downarrow c \\
  & c \leq 0 \\
  & \Rightarrow H ; \text{if } e \ s_1 \rightarrow H ; s_2 \\
  \text{WHILE} & H ; \text{while } e \ s \rightarrow H ; \text{if } e (s ; \text{while } e) \text{ skip}
\end{align*}
\]

Call-by-value, left-to-right evaluation language presented in class:

\[
\begin{align*}
  e & ::= \lambda x. e \mid x \mid e \mid e + c \\
  v & ::= \lambda x. e \mid c
\end{align*}
\]

\[
\begin{align*}
  (\lambda x. e) v \rightarrow e[v/x] \\
  e \rightarrow e'
\end{align*}
\]

\[
\begin{align*}
  e[v/x] = e''
\end{align*}
\]

\[
\begin{align*}
  x[e/x] = e \\
  y \neq x \\
  y[e/x] = y
\end{align*}
\]

\[
\begin{align*}
  e_1[e/x] = e_1' \\
  e_2[e/x] = e_2'
\end{align*}
\]

\[
\begin{align*}
  (\lambda y. e_1)[e/x] = \lambda y. e_1'
\end{align*}
\]

\[
\begin{align*}
  & \Gamma \vdash e : \tau \\
  & \Gamma \vdash c : \text{int} \\
  & \Gamma \vdash x : \Gamma(x) \\
  \Gamma, x : \tau_1 \vdash e : \tau_2 \\
  \Gamma \vdash \lambda x. e : \tau_1 \rightarrow \tau_2 \\
  \Gamma \vdash e_1 : \tau_1 \\
  \Gamma \vdash e_2 : \tau_2 \\
  \Gamma \vdash e_1 \ e_2 : \tau_2
\end{align*}
\]

- If \(\vdash e : \tau\) and \(e \rightarrow e'\), then \(\vdash e' : \tau\).
- If \(\vdash e : \tau\), then \(e\) is a value or there exists an \(e'\) such that \(e \rightarrow e'\).
- If \(\Gamma, x : \tau' \vdash e : \tau\) and \(\Gamma \vdash e' : \tau'\), then \(\Gamma \vdash e'[x]/\tau\).
1. (20 points) OCaml and functional programming. Note there is a part (a) and part (b) to this problem.

(a) For each OCaml function below (q1, q2, and q3):

- Describe in 1–2 English sentences what the function computes.
- Give the type of the function. (Hint: For all three functions, the type has one type variable.)
  Recall that function types are of the form \( t_1 \rightarrow t_2 \rightarrow \ldots \rightarrow t_n \) (where \( t_1, \ldots, t_{n-1} \) are the types of the arguments and \( t_n \) is the type of the value returned by the function).

```ocaml
let q1 x =  
  let rec g x y = 
    match x with 
    | [] -> y 
    | hd::tl -> g tl (hd::y) 
  in g x []

let rec q2 f lst =  
  match lst with 
  | [] -> [] 
  | hd::tl -> if f hd then hd::(q2 f tl) 
    else q2 f tl

let q3 x g = g (g x)
```

(b) Consider this purposely complicated code that uses q3 as defined above.

```ocaml
let x = q3 2  
let y z = z+z  
let z = 9  
let x = x y
```

After evaluating this code, what is \( x \) bound to (i.e., what is the value of \( x \))?

Solution:

(a) • q1 takes a list and returns its reverse. It has type \('a list -> 'a list\).
• q2 takes a function and a list and returns the list containing all the elements from the input list (in order) for which the function applied to the element returns true. (It's a filter.) It has type \('a -> bool) -> 'a list -> 'a list\).
• q3 returns the result of applying its second argument to the result of applying its second argument to its first argument. It has type \('a -> ('a -> 'a) -> 'a\).

(b) 8
2. (15 points) IMP with large- and small-step semantics.

(a) For each of the following IMP programs, will evaluation terminate? If yes, write a derivation using the appropriate large- or small-step semantics rules for each program (recall that the general form of each step of the derivation is \( H_1 : s_1 \rightarrow H_2 : s_2 \)). Indicate the names of rule(s) you are applying at each step. If the evaluation does not terminate, briefly explain why.

i. \( \{ x \rightarrow 2, z \rightarrow 3 \}; \text{if (x) (x := x + (-4)) (skip); x := x \times z} \)

ii. \( \{ x \rightarrow 2, y \rightarrow 0 \}; (\text{while (x) y := y + x); x := x + (-1)} \)

(b) Does the following program terminate? If no, briefly explain why. If yes, without showing the complete derivation, what is the final value of \( \text{ans} \)?

\[
x := 5;
\text{ans} := 1;
\text{while(x) (}
\text{ans} := \text{ans} + x;
\text{x := x + (-1)}
\text{)}
\]

Solution:

(a) i. \( \{ x \rightarrow 2, z \rightarrow 3 \}; \text{if (x) (x := x + (-4)) (skip); x := x \times z} \)

ii. \( \{ x \rightarrow 2, y \rightarrow 0 \}; (\text{while (x) y := y + x); x := x + (-1)} \)

(b) It terminates, \( \text{ans} \) is 16.
3. (36 points) (IMP with toggle)
This problem adds a single toggle to IMP. The toggle has two states: up and down. A new expression form \texttt{read} evaluates to 1 if the toggle is currently up and 0 if the toggle is currently down. A new statement form \texttt{toggle} switches the state of the toggle. The judgment forms for the operational semantics are adapted accordingly.

\begin{align*}
e & ::= \ldots | \texttt{read} \\
s & ::= \ldots | \texttt{toggle} \\
t & ::= \texttt{up} | \texttt{down}
\end{align*}

(a) (6 points) Give all the inference rules for large-step expression evaluation (Hint: 6 rules).

(b) (8 points) Give all the inference rules for small-step statement evaluation (Hint: 8 rules).

(c) (18 points) If this statement is true, prove it formally, else give a counterexample:
If \( H; \texttt{up}; t \rightarrow H' \); \( t \rightarrow H' \); \( t \rightarrow s \rightarrow H' \); \( t \rightarrow \texttt{read} \) where \( s' \) is \( s \) with every \texttt{read} replaced by 1.

\begin{align*}
& (a) \\
& H; t; c \Downarrow c & H; t; x \Downarrow H(x) & H; t; c_1 \Downarrow c_1 & H; t; c_2 \Downarrow c_2 \\
& H; t; e_1 \Downarrow c_1 & H; t; e_2 \Downarrow c_2 & \downarrow c \leq 0 & \downarrow c > 0 \\
& H; t; e \Downarrow c & \downarrow c \leq 0 & \downarrow c > 0
\end{align*}

(d) (4 points) If this statement is true, prove it formally, else give a counterexample:
(Notice the \( \ast \) for 0 or more steps)
If \( H; \texttt{up}; s \rightarrow^\ast H'; \texttt{up}; \texttt{skip} \), then \( H; \texttt{up}; s' \rightarrow^\ast H'; \texttt{up}; \texttt{skip} \) where \( s' \) is \( s \) with every \texttt{read} (in every expression) replaced by 1.

Solution:

(a)

(b)

(c) see next page

(d) see next page
(Extra space for answering problem 3)

**Solution:**

(c) This statement is true. We prove it by induction on the derivation of \( H; \text{up}; e \Downarrow c \), proceeding by cases on the bottommost rule in the derivation:

- If \( e \) is a constant, then \( e' = e \) so the assumed derivation is the derivation we need.
- If \( e \) is a variable, then \( e' = e \) so the assumed derivation is the derivation we need.
- If \( e \) is \( e_1 + e_2 \) for some \( e_1 \) and \( e_2 \), then \( H; \text{up}; e \Downarrow c \) \( H; \text{up}; e_2 \Downarrow c_2 \) where \( c = c_1 + c_2 \).
  
  So by induction \( H; \text{up}; e_1 \Downarrow c_1 \) \( H; \text{up}; e_2 \Downarrow c_2 \) where \( e_1' \) and \( e_2' \) are \( e_1 \) and \( e_2 \) with \text{read} replaced by 1. So we can use the rule for addition to derive \( H; \text{up}; e_1' + e_2 \Downarrow c_1 + c_2 \).
  
  This is what we need because \( e_1' + e_2' \) is \( e \) with \text{read} replaced by 1 and \( c = c_1 + c_2 \).
- If \( e \) is \( e_1 * e_2 \) for some \( e_1 \) and \( e_2 \), then \( H; \text{up}; e \Downarrow c \) \( H; \text{up}; e_2 \Downarrow c_2 \) where \( c = c_1 * c_2 \).
  
  So by induction \( H; \text{up}; e_1 \Downarrow c_1 \) \( H; \text{up}; e_2 \Downarrow c_2 \) where \( e_1' \) and \( e_2' \) are \( e_1 \) and \( e_2 \) with \text{read} replaced by 1. So we can use the rule for multiplication to derive \( H; \text{up}; e_1' * e_2 \Downarrow c_1 * c_2 \).
  
  This is what we need because \( e_1' * e_2' \) is \( e \) with \text{read} replaced by 1 and \( c = c_1 * c_2 \).
- If \( e \) is \text{read} and the toggle is \text{up}, then \( c = 1 \) and \( e' = 1 \) and we can use the rule for constants to derive \( H; \text{up}; e \Downarrow 1 \).
- The rule where \( e \) is \text{read} and the toggle is \text{down} cannot end the derivation of \( H; \text{up}; e \Downarrow c \), so this case holds vacuously.

(d) This statement is false. There are an infinite number of countexamples, such as:

\[ \text{.; up; toggle; (x := read; toggle) } \rightarrow^{*} \text{., x } \rightarrow 0; \text{up; skip}, \]

but

\[ \text{.; up; toggle; (x := 1; toggle) } \rightarrow^{*} \text{., x } \rightarrow 1; \text{up; skip,} \]
4. (14 points) In this problem, we use the untyped lambda calculus with small-step call-by-value left-to-right evaluation (recall that in call-by-value, function application evaluates the argument before it proceeds to the evaluation of the function’s body).

Recall this encoding of pairs:

- “mkpair” λx. λy. λz. z x y
- “fst” λp. p λx. λy. x
- “snd” λp. p λx. λy. y

We would expect a correct encoding to show “fst” (“mkpair” z z) evaluates to z. But this sequence of steps allegedly shows that “fst” (“mkpair” z z) evaluates to “fst”:

(a) The sequence of steps is wrong. Which steps are wrong and why are they wrong?

(b) Show a correct sequence of steps that produces z but is otherwise very similar to the sequence of steps shown above.

Solution:

(a) The first two steps both capture z. We should α-convert λz. z x y in order to perform these first two steps properly.

(b)
5. (15 points) In this problem, assume the simply-typed lambda calculus with constants. For each of the following:

- If the answer is yes, give an example $\Gamma$ and $\tau$.
- If the answer is no, you can just say “no.”

(a) Is there a $\Gamma$ and $\tau$ such that $\Gamma \vdash (\lambda x. x) \ x : \tau$?
(b) Is there a $\Gamma$ and $\tau$ such that $\Gamma \vdash \lambda x. (x \ x) : \tau$?
(c) Is there a $\Gamma$ and $\tau$ such that $\Gamma \vdash x \ x : \tau$?
(d) Is there a $\Gamma$ and $\tau$ such that $\Gamma \vdash x \ (\lambda x. x) : \tau$?
(e) Is there a $\Gamma$ and $\tau$ such that $\Gamma \vdash c \ x : \tau$?

Solution:

(a) Yes, for example $\Gamma = \cdot, x: \text{int}$ and $\tau = \text{int}$. In general, the type of $x$ in $\Gamma$ has to be $\tau$.
(b) No
(c) No
(d) Yes, for example $\Gamma = \cdot, x: \text{(int} \rightarrow \text{int}) \rightarrow \text{int}$ and $\tau = \text{int}$. In general, the type of $x$ in $\Gamma$ has to have the form $(\tau' \rightarrow \tau') \rightarrow \tau$.
(e) No
6. OPTIONAL, extra credit (+6 points) Consider this lemma, which is slightly different from the Preservation Lemma we proved for the simply-typed lambda calculus:

Differently Preserved: If $\cdot \vdash e : \tau$ and $e \rightarrow e'$, then there exists a $\tau'$ such that $\cdot \vdash e' : \tau'$.

(a) Is the Differently Preserved Lemma weaker, stronger, or incomparable to the Preservation Lemma? Explain.
(b) Is the Differently Preserved Lemma true? Explain.
(c) Is the Differently Preserved Lemma (instead of the Preservation Lemma) and the Progress Lemma sufficient to prove Type Safety? Explain.

Solution:

(a) It is weaker: The Preservation Lemma implies the Differently Preserved Lemma just by choosing $\tau'$ to be $\tau$. (Something is weaker than something else that implies it.)
(b) Yes, the Preservation Lemma is true and it implies the Differently Preserved Lemma.
(c) Yes, just like the Preservation Lemma, the Differently Preserved Lemma and induction on the number of steps taken ensure that no well-typed program can become ill-typed. And Progress ensures no well-typed program is stuck.