CIS 624, Fall 2016, Midterm Examination
7 November 2016

Please do not turn the page until everyone is ready.

Rules:

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

• Please stop promptly at 15:20.

• 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>45</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>12</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>8 (+4 extra)</td>
<td></td>
</tr>
<tr>
<td>Total</td>
<td>100 (+4 extra)</td>
<td></td>
</tr>
</tbody>
</table>
For your reference:

IMP small-step semantics as presented in class:

\[
\begin{align*}
\text{const} & \quad \text{IMP syntax as presented in class:} \\
H & \quad \text{IMP large-step semantics as presented in class:}
\end{align*}
\]

IMP small-step semantics as presented in class:

\[
\begin{align*}
\text{assign} & \quad \text{seq1} \\
\text{while} & \quad \text{seq2}
\end{align*}
\]

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

\[
\begin{align*}
e & ::= \lambda x. e | x | e \ | \ c \\
v & ::= \lambda x. e | c
\end{align*}
\]
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) (10 points) 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. \( \{ x \rightarrow 2, z \rightarrow 3 \}; \text{if} (x) (x := x + (-4)) (\text{skip}); x := x \cdot z \)

\( \rightarrow \{ x \rightarrow 2, z \rightarrow 3 \}; x := x + (-4); x := x \cdot z \) by \[\text{SEQ2}] [\text{VAR}] [\text{IF1}]

\( \rightarrow 5 \{ x \rightarrow 2, z \rightarrow 3, x \rightarrow -2 \}; \text{skip}; x := x \cdot z \) by \[\text{ASSIGN}] [\text{VAR}] [\text{CONST}] [\text{ADD}]

\( \rightarrow 2 \{ x \rightarrow 2, z \rightarrow 3, x \rightarrow -2, x \rightarrow -6 \}; \text{skip} \) by \[\text{SEQ1}] [\text{MULT}] [\text{ASSIGN}] \) Hence, terminates.

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

(b) (5 points) Does the following program terminate? If no, briefly explain why. If yes, without showing the complete derivation, what is the final value of \( \text{ans} \)?

\[ \begin{align*}
x &:= 5; \\
\text{ans} &:= 1; \\
\text{while}(x) \ (
\text{ans} &:= \text{ans} + x; \\
x &:= x + (-1)
\end{align*} \)

Solution:

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

\( \rightarrow \{ x \rightarrow 2, z \rightarrow 3 \}; x := x + (-4); x := x \cdot z \) by \[\text{SEQ2}] [\text{VAR}] [\text{IF1}]

ii. Does not terminate, the value of \( x \) is never decremented in the body of the while, so we have to keep applying the WHILE rule infinitely many times.

(b) It terminates, \( \text{ans} \) is 16.
3. (44 points) (IMP with toggle)
This problem adds a single toggle to IMP. The toggle has two states: up and down. A new expression form read evaluates to 1 if the toggle is currently up and 0 if the toggle is currently down. A new statement form toggle switches the state of the toggle. The judgment forms for the operational semantics are adapted accordingly.

\[
e ::= \ldots \mid \text{read} \\
s ::= \ldots \mid \text{toggle} \\
t ::= \text{up} \mid \text{down}
\]

(a) (12 points) Give all the inference rules for large-step expression evaluation (Hint: 6 rules).

(b) (16 points) Give all the inference rules for small-step statement evaluation (Hint: 8 rules).

(c) (12 points) If this statement is true, prove it formally, else give a counterexample:
   If \( H; \text{up}; e \Downarrow c \), then \( H; \text{up}; e' \Downarrow c \) where \( e' \) is \( e \) with every read replaced by 1.

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

Solution:

(a)

\[
\frac{H; t; c \Downarrow c}{H; t; \Downarrow H(x)} \\
\frac{H; t; e_1 \Downarrow c_1}{H; t; e_1 + e_2 \Downarrow c_1 + c_2} \\
\frac{H; t; e_1 \Downarrow c_1 \quad H; t; e_2 \Downarrow c_2}{H; t; e_1 * e_2 \Downarrow c_1 * c_2} \\
\frac{H; t; e_1 \Downarrow c_1 \quad H; t; e_2 \Downarrow c_2}{H; t; e \Downarrow c_c \quad H; t; e \Downarrow c_c \quad H; t; e \Downarrow c \quad c > 0}
\]

(b)

\[
\frac{H; t; e \Downarrow c}{H; t; x := c \rightarrow H, x = c \Downarrow t; \text{skip}} \\
\frac{H; t; s_1 \rightarrow H'; t'; s'_1}{H; t; \text{while } e \Downarrow H; t; \text{if } e \Downarrow s_1 \text{ while } e \Downarrow H; t; s_1}
\]

(c) see next page

(d) see next page
(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 $H ; \text{up} ; e_1 \Downarrow c_1$ and $H ; \text{up} ; e_2 \Downarrow c_2$ where $c = c_1 + c_2$.
  So by induction $H ; \text{up} ; e_1' \Downarrow c_1$ and $H ; \text{up} ; e_2' \Downarrow c_2$ 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 $H ; \text{up} ; e_1 \Downarrow c_1$ and $H ; \text{up} ; e_2 \Downarrow c_2$ where $c = c_1 * c_2$.
  So by induction $H ; \text{up} ; e_1' \Downarrow c_1$ and $H ; \text{up} ; e_2' \Downarrow c_2$ 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 up, then $c$ is 1 and $e'$ is 1 and we can use the rule for constants to derive $H ; \text{up} ; 1 \Downarrow 1$.
- The rule where $e$ is $\text{read}$ and the toggle is 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:

. ; up ; toggle; ($x := \text{read}; \text{toggle}$) $\Rightarrow^* ., x \mapsto 0 ;$ up; skip,

but

. ; up ; toggle; ($x := 1; \text{toggle}$) $\Rightarrow^* ., x \mapsto 1 ;$ up; skip
4. (12 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” \( \lambda x. \lambda y. \lambda z. z \times y \)
- “fst” \( \lambda p. p \lambda x. \lambda y. x \)
- “snd” \( \lambda p. p \lambda x. \lambda y. y \)

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

\[
(\lambda p. p \lambda x. \lambda y. x)((\lambda x. \lambda y. \lambda z. z \times y) z z) \\
\rightarrow (\lambda p. p \lambda x. \lambda y. x)((\lambda y. \lambda z. z z y) z) \\
\rightarrow (\lambda p. p \lambda x. \lambda y. x)(\lambda z. z z z) \\
\rightarrow (\lambda p. p \lambda x. \lambda y. x)(\lambda z. \times \times z) \\
\rightarrow (\lambda x. \lambda y. x)(\lambda x. \lambda y. x)(\lambda x. \lambda y. x) \\
\rightarrow (\lambda y. (\lambda x. \lambda y. x))(\lambda x. \lambda y. x) \\
\rightarrow (\lambda x. \lambda y. x)
\]

(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 \( \alpha \)-convert \( \lambda z. z \times y \) in order to perform these first two steps properly.

(b)
5. (8 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) (2 points) Is there a $\Gamma$ and $\tau$ such that $\Gamma \vdash (\lambda x. x) \ x : \tau$?
(b) (2 points) Is there a $\Gamma$ and $\tau$ such that $\Gamma \vdash \lambda x. (x \ x) : \tau$?
(c) (2 points) Is there a $\Gamma$ and $\tau$ such that $\Gamma \vdash \ x \ x : \tau$?
(d) (2 points) Is there a $\Gamma$ and $\tau$ such that $\Gamma \vdash c \ x : \tau$?
(e) Extra credit (+4 pts) Is there a $\Gamma$ and $\tau$ such that $\Gamma \vdash x \ (\lambda x. x) : \tau$?

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) No
(e) Yes, for example $\Gamma = \cdot, x: (\text{int} \to \text{int}) \to \text{int}$ and $\tau = \text{int}$. In general, the type of $x$ in $\Gamma$ has to have the form $(\tau' \to \tau') \to \tau$. 