CIS 624, Fall 2017, Midterm Examination
7 November 2017

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>25</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>15</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>40</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 syntax as presented in class:

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

IMP large-step semantics as presented in class:

\[
\begin{align*}
  H; e \Downarrow c \\
  \text{CONST} & \quad \text{VAR} & \quad \text{ADD} & \quad \text{MULT} \\
  H; c \Downarrow c & \quad H; x \Downarrow H(x) & \frac{H; c_1 \Downarrow c_1}{H; c_2 \Downarrow c_2} & \frac{H; c_1 \Downarrow c_1}{H; c_2 \Downarrow c_2}
\end{align*}
\]

IMP small-step semantics as presented in class:

\[
\begin{align*}
  H_1; s_1 \rightarrow H_2; s_2 \\
  \text{ASSIGN} & \quad \text{SEQ1} & \quad \text{SEQ2} \\
  H; e \Downarrow c & \quad H; \text{skip} ; s \rightarrow H; s & \frac{H; s_1 \rightarrow H'; s_1' \quad s \rightarrow H; s_2}{H; s_2 \rightarrow H'; s_2'}
\end{align*}
\]

If \(H; e \Downarrow c\) and \(c > 0\):

\[
H; \text{if } e \text{ s } s \rightarrow H; s_1
\]

If \(H; e \Downarrow c\) and \(c \leq 0\):

\[
H; \text{while } e s \rightarrow H; s_2
\]

\[
H(x) = \begin{cases} 
  c & \text{if } H = H', x \mapsto c \\
  H'(x) & \text{if } H = H', y \mapsto c' \text{ and } y \neq x \\
  0 & \text{if } H = \cdot
\end{cases}
\]

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

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

\[
\begin{align*}
  e \rightarrow e' \\
  \frac{(\lambda x. e) v \rightarrow e[v/x]}{e \rightarrow e'} & \quad \frac{e_1 \rightarrow e'_1}{e_1 e_2 \rightarrow e'_1 e_2} & \quad \frac{e_2 \rightarrow e'_2}{v e_2 \rightarrow v e'_2}
\end{align*}
\]

\[
\begin{align*}
  e[e'/x] = e'' \\
  \frac{e[e'/x] = e''}{x[e/x] = e} & \quad \frac{y \neq x}{y[e/x] = y} & \quad \frac{e_1[e/x] = e'_1}{(\lambda y. e_1)[e/x] = \lambda y. e'_1}
\end{align*}
\]

\[
\begin{align*}
  \frac{e_1[e/x] = e'_1 \quad y \neq x \quad y \not\in \text{FV}(e)}{e_1[e/x] = e'_1} & \quad \frac{e_2[e/x] = e'_2}{(e_1 e_2)[e/x] = e'_1 e'_2}
\end{align*}
\]

\[
\begin{align*}
  \Gamma \vdash e : \tau \\
  \Gamma \vdash e : \tau_1 & \quad \Gamma \vdash e : \tau_2 & \quad \Gamma \vdash e : \tau_2 \rightarrow \tau_1 & \quad \Gamma \vdash e : \tau_1
\end{align*}
\]
1. (25 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. Recall that function types are of the form t₁ → t₂ → ... → tₙ
  (where t₁, .., tₙ₋₁ are the OCaml types of the arguments and tₙ is the OCaml type of the value returned by the function).

\[
\text{let rec q1 x lst = (\*) Write the type of q1 below: *)}
\]
\[
\begin{array}{c}
\text{match lst with} \\
[\] \rightarrow [x] \\
| \text{h :: t} \rightarrow \\
\quad \text{if } x \leq \text{h} \text{ then } x :: \text{lst} \\
\quad \text{else } \text{h :: q1 x t}
\end{array}
\]

\[
\text{let rec q2 lst = (\*) Write the type of q2 below: *)}
\]
\[
\begin{array}{c}
\text{match lst with} \\
[\] \rightarrow [\] \\
| \text{head :: tail} \rightarrow \text{q1 head (q2 tail)}
\end{array}
\]

\[
\text{let rec q3 x y = (\*) Write the type of q3 below: *)}
\]
\[
\begin{array}{c}
\text{match x, y with} \\
[\],_ \rightarrow y \\
| _,[\] \rightarrow x \\
| \text{hx :: tx, hy :: ty} \rightarrow \\
\quad \text{if } \text{hx} < \text{hy} \text{ then } \text{hx :: q3 tx y} \\
\quad \text{else } \text{hy :: q3 x ty}
\end{array}
\]

(b) Consider this code that uses q2 and q3 as defined above.

\[
\text{let a = q2 [6;2;-1;4]} \\
\text{let b = q2 [5;2;7]} \\
\text{let c = q3 a b}
\]

After evaluating this code, what are the values of the variables a, b, and c?

Solution:

(a) • q1 takes a (sorted) list and a value x, then inserts x, returning a sorted list containing x. It has type 'a → 'a list → 'a list.
• q2 takes a list and returns the sorted list. argument. It has type 'a list → 'a list.
• q3 takes two sorted lists, merges them, and returns the resulting sorted list. It has type 'a list → 'a list → 'a list.

(b) a is [-1; 2; 4; 6], b is [2; 5; 7], c is [-1; 2; 2; 4; 5; 6; 7]
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. $\{x \rightarrow 2, z \rightarrow 2\}; x := x + (-1); \text{while}(x+z) z := z + x$

ii. $\{x \rightarrow 0, z \rightarrow 2\}; \text{if}(x) (x := x + 1) (\text{skip}); x := x + z$

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

\[
x := 2;
while(x) (\text{\ }
\quad \text{ans} := \text{ans} + x; \\
\quad x := x + (-1)
\text{\ })
\]

Solution:

(a) i. 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.

ii. $\{x \rightarrow 0, z \rightarrow 2\}; \text{if}(x) (x := x + 1) (\text{skip}); x := x + z$

$\rightarrow^3 \{x \rightarrow 0, z \rightarrow 2\}; \text{skip}; x := x + z$ by $[\text{SEQ2}][\text{IF2}][\text{VAR}]$

$\rightarrow \{x \rightarrow 0, z \rightarrow 2\}; x := x + z$ by $[\text{SEQ1}]$

$\rightarrow^4 \{x \rightarrow 0, z \rightarrow 2, x \rightarrow 2\}; \text{skip}$ by $[\text{ASSIGN}][\text{ADD}][\text{VAR}][\text{VAR}]$ Hence, terminates.

(b) It terminates, ans is 3.
3. (40 points) This problem considers a language that is like the language for IMP expressions (not statements), but where we have pixels instead of integers. A pixel value contains three numbers between 0 and 255 (the first for red, the second for green, the third for blue, but that is not relevant much). Here is the syntax and an English description of the semantics:

$$\begin{align*}
e & ::= p \mid x \mid e + e \mid \text{lighten } e \mid \text{darken } e \\
p & ::= (c, c, c) \\
H & ::= \cdot \mid H, x \mapsto p \\
(c & \in \{0, 1, \ldots, 255\}) \\
(x & \in \{x_1, x_2, \ldots, y_1, y_2, \ldots, z_1, z_2, \ldots\})
\end{align*}$$

- Heaps and variables work as usual, with values being pixels.
- As indicated in the syntax, all parts of pixel values must always be between 0 and 255 inclusive.
- Addition adds each component of its pixel arguments separately to produce a new pixel, with sums greater than 255 “rounded down” to 255.
- A “lighten” expression produces a pixel with each component being one more than it was in the argument, again with a max of 255 (i.e., 255 stays 255).
- A “darken” expression produces a pixel with each component being one less than it was in the argument, with a min of 0 (i.e., 0 stays 0).

(Notice higher values are lighter.)

(a) (12 pts) Give a large-step operational semantics for this language, with a judgment of the form $H ; e \Downarrow p$.

- Hint: Use 5 rules. This is a good hint: there are other approaches that need many more rules. However, you will not lose points if you use more rules that fully define the semantics.
- Assume that you can use from mathematics (“blue math” in terms of lecture) the following operations: $\min(x, y)$ and $\max(x, y)$ for computing the minimum and maximum of two numbers. Also use addition and subtraction.

(b) (16 pts) Using your answer to part (a), prove this: $H ; \text{lighten } e \Downarrow p$ if and only if $H ; e + (1, 1, 1) \Downarrow p$. Make sure to prove both directions.

(c) (8 pts) Define inference rules for a predicate $\text{noblack}(e)$ that holds if none of the pixel constants in $e$ are the constant $(0, 0, 0)$ and $e$ contains no “darken” expressions.

For example, $\text{noblack}(\text{lighten } (1, 123, 0))$ $\text{noblack}((1, 2, 3))$ do hold, but $\text{noblack}(\text{darken } (1, 123, 0))$ and $\text{noblack}((0, 0, 0) + (1, 1, 1))$ do not hold.

You can assume that the comparison operators $=$ and $\neq$ can be used with pixels.

(d) (4 pts) Disprove this: If $H ; e \Downarrow p$ and $\text{noblack}(e)$, then $p \neq (0, 0, 0)$. 


Solution:

(a) \[
\begin{align*}
H &: p \downarrow p & H &: x \downarrow H(x) \\
H &: e_1 \downarrow \langle c_1, c_2, c_3 \rangle & H &: e_2 \downarrow \langle c_4, c_5, c_6 \rangle \\
H &: e_1 + e_2 \downarrow \langle \min(255, c_1 + c_4), \min(255, c_2 + c_5), \min(255, c_3 + c_6) \rangle \\
H &: e \downarrow \langle c_1, c_2, c_3 \rangle \\
H &: \text{lighten } e \downarrow \langle \min(255, c_1 + 1), \min(255, c_2 + 1), \min(255, c_3 + 1) \rangle \\
H &: \text{darken } e \downarrow \langle \max(0, c_1 - 1), \max(0, c_2 - 1), \max(0, c_3 - 1) \rangle
\end{align*}
\]

(b) Prove the two directions separately. First assume \(H \downarrow \text{lighten } e \downarrow p\).
Inversion (only the lighten rule applies) ensures there is some \(c_1, c_2, \) and \(c_3\) such that \(p = \langle \min(255, c_1 + 1), \min(255, c_2 + 1), \min(255, c_3 + 1) \rangle\) and \(H \downarrow e \downarrow \langle c_1, c_2, c_3 \rangle\). So we can use \(H \downarrow e \downarrow \langle c_1, c_2, c_3 \rangle\) and the addition rule to derive:

\[
\begin{align*}
H &: e \downarrow \langle c_1, c_2, c_3 \rangle & H &: \langle 1, 1, 1 \rangle \downarrow \langle 1, 1, 1 \rangle \\
H &: e + \langle 1, 1, 1 \rangle \downarrow p
\end{align*}
\]

Now assume \(H \downarrow e + \langle 1, 1, 1 \rangle \downarrow p\). Then inversion (only the addition rule applies) ensures there is some \(c_1, c_2, \) and \(c_3\) such that \(p = \langle \min(255, c_1 + 1), \min(255, c_2 + 1), \min(255, c_3 + 1) \rangle\) and \(H \downarrow e \downarrow \langle c_1, c_2, c_3 \rangle\) (because another inversion ensures \(\langle 1, 1, 1 \rangle\) can evaluate only to itself). So we can use \(H \downarrow e \downarrow \langle c_1, c_2, c_3 \rangle\) and the lighten rule to derive:

\[
\begin{align*}
H &: e \downarrow \langle c_1, c_2, c_3 \rangle \\
H &: \text{lighten } e \downarrow p
\end{align*}
\]

(c) (Note if you insist that \(=\) and \(\neq\) be used only on integers, then we need three rules for pixel constants.)

<table>
<thead>
<tr>
<th>noblack(x)</th>
<th>noblack(p)</th>
<th>noblack(e_1)</th>
<th>noblack(e_2)</th>
<th>noblack(e)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(p \neq (0,0,0))</td>
<td>(\text{noblack}(e_1))</td>
<td>(\text{noblack}(e_2))</td>
<td>(\text{noblack}(e))</td>
<td></td>
</tr>
</tbody>
</table>

(d) This is false because of variables. Consider any heap \(H\) where \(H(x) = (0,0,0)\). Then \(H \downarrow x \downarrow (0,0,0)\) but \(\text{noblack}(x)\).
Name:__________________________________________

(More room for answering problem 3)
4. **(12 points)** In this problem, we use the untyped lambda calculus with small-step call-by-value left-to-right evaluation

**Given**

- true: \( \lambda x. \lambda y. x \)
- false: \( \lambda x. \lambda y. y \)
- not: \( \lambda x. ((x \text{ false}) \text{ true}) \)

Prove that \( (\text{not} \ (\text{not} \ \text{true})) \) reduces to \( \text{true} \) using this encoding. Show the complete sequence of steps (you can choose to use the shorthand mnemonics above as much as possible). Write a brief explanation at each step of what you are doing in that step, e.g., replacing “not” with encoding, beta reduction, alpha conversion, etc..

**Solution:**

\[
\begin{align*}
\text{not (not true)} & \quad \text{// replacing 2nd not w/ encoding} \\
\rightarrow & \quad \text{not (\lambda x. ((x \text{ false}) \text{ true}) \text{ true})} \quad \text{// beta-reduction: [true/x]} \\
\rightarrow & \quad \text{not (((true \text{ false}) \text{ true}) \text{ true})} \quad \text{// replacing 1st true w/ encoding} \\
\rightarrow & \quad \text{not (((\lambda x. \lambda y. x) \text{ false}) \text{ true})} \quad \text{// beta-reduction: [false/x]} \\
\rightarrow & \quad \text{not (((\lambda y. \text{ false}) \text{ true})} \quad \text{// expand 1st false and do alpha-conversion of y} \\
\rightarrow & \quad \text{not (((\lambda p. \lambda x. \lambda y. y) \text{ true}) \text{ true})} \quad \text{// beta-reduction: [true/p]} \\
\rightarrow & \quad \text{not (\lambda x. \lambda y. y)} \quad \text{// replacing not with encoding} \\
\rightarrow & \quad (\lambda x. ((x \text{ false}) \text{ true}))(\lambda x. \lambda y. y) \quad \text{// beta-reduction: [(\lambda x. \lambda y. y)/x]} \\
\rightarrow & \quad ((\lambda x. \lambda y. y) \text{ false}) \text{ true} \quad \text{// beta-reduction: [false/x]} \\
\rightarrow & \quad (\lambda y. y) \text{ true} \quad \text{// beta-reduction: [true/y]} \\
\rightarrow & \quad \text{true}
\end{align*}
\]
5. (12 points) In this problem, assume the simply-typed lambda calculus with constants. For each of the following:

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

(a) Is there a $\Gamma$ and $\tau$ such that $\Gamma \vdash \lambda y. (x \ y) : \tau$?
(b) Is there a $\Gamma$ and $\tau$ such that $\Gamma \vdash v \ v : \tau$ (where $v$ is a value)?
(c) Is there a $\Gamma$ and $\tau$ such that $\Gamma \vdash (\lambda x. \lambda y. \ x \ y) \ z : \tau$?
(d) Is there a $\Gamma$ and $\tau$ such that $\Gamma \vdash v \ c : \tau$ (where $v$ is a value and $c$ is a constant)?
(e) Extra credit (+4 pts) Is there a $\Gamma$ and $\tau$ such that $\Gamma \vdash z \ (\lambda x. \ x) \ (\lambda y. \ y) : \tau$?

\textbf{Solution:}

(a) Yes, for example $\Gamma = \cdot, x: \tau_1 \to \tau_2$ and $\tau = \tau_1 \to \tau_2$
(b) No
(c) Yes, for example $\Gamma = \cdot, z: \tau_1 \to \tau_2$ and $\tau = \tau_1 \to \tau_2$ (application typing rule).
(d) Yes, $\Gamma = \cdot, v: \text{int} \to \text{int}$ and $\tau = \text{int}$.
(e) Yes, for example $\Gamma = \cdot, z: (\tau_1 \to \tau_1) \to (\tau_2 \to \tau_2) \to \tau_3$ and $\tau = \tau_3$. 
Name:_____________________________________

(This page intentionally blank, use as you wish.)