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.
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 \( v ::= c \mid \text{error} \). Make sure that if evaluation of \( e \) under \( H \) would involve dividing by 0, then \( H; e \Downarrow \text{error} \).

(Hint: You need 3 more inference rules, so 6 total.)

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

Solution:

(a)

\[
\begin{align*}
H; c \Downarrow c \\
H; x \Downarrow H(x) \\
H; e_1/e_2 \Downarrow c_1/c_2 \quad & \quad c_2 \neq 0
\end{align*}
\]

(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:

\[
\begin{align*}
H; e_1 \Downarrow \text{error} \\
H; e_1/e_2 \Downarrow \text{error} \\
H; e_2 \Downarrow \text{error} \\
H; e_1/e_2 \Downarrow \text{error} \\
H; e_2 \Downarrow 0 \\
H; e_1/e_2 \Downarrow \text{error}
\end{align*}
\]

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

\[
s ::= \text{skip} \mid x := e \mid s ; s \mid \text{if } e \text{ s} \text{ s} \mid \text{while } e \text{ s}
\]

\[
\begin{array}{c}
\text{ASSIGN} \\
H ; e \Downarrow c \\
\hline
H ; x := e \rightarrow H, x \mapsto c ; \text{skip}
\end{array}
\begin{array}{c}
\text{SEQ1} \\
H ; \text{skip} \rightarrow H ; s \\
\hline
H ; s_1 ; s_2 \rightarrow H' ; s_1' ; s_2
\end{array}
\begin{array}{c}
\text{SEQ2} \\
H ; s_1 \rightarrow H' ; s_1' \\
\hline
H ; \text{if } e \text{ s}_1 \text{ s}_2 \rightarrow H ; s_2
\end{array}
\begin{array}{c}
\text{IF1} \\
H ; e \Downarrow c \quad c > 0 \\
\hline
H ; \text{if } e \text{ s}_1 \text{ s}_2 \rightarrow H ; s_1
\end{array}
\begin{array}{c}
\text{IF2} \\
H ; e \Downarrow c \quad c \leq 0 \\
\hline
H ; \text{while } e \text{ s}_1 \text{ s}_2 \rightarrow H ; s_2
\end{array}
\begin{array}{c}
\text{WHILE} \\
H ; \text{while } e \text{ s}_1 \text{ s}_2 \rightarrow H ; s_2
\end{array}
\]

(a) Define a judgment of the form 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 mysize(skip; 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 → H' ; s' and mysize(s) = n and 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)

\[
\begin{array}{c}
\text{mysize(skip)} = 1 \\
\text{mysize(x := e)} = 2 \\
\text{mysize(s_1)} = n_1 \\
\text{mysize(s_2)} = n_2
\end{array}
\begin{array}{c}
\text{mysize(s_1 ; s_2)} = n_1 + n_2 \\
\text{mysize(if } e \text{ s}_1 \text{ s}_2) = n_1 + n_2 \\
\text{mysize(while } e \text{ s)} = n
\end{array}
\]

(b) The proof is by induction on the derivation of H ; s → 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 → H' ; s'_1. Furthermore, mysize(s_1; s_2) = n_1 + n_2 where mysize(s_1) = n_1 and mysize(s_2) = n_2. So by induction mysize(s'_1) = n'_1 where n'_1 < n_1. So we can derive 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 mysize(s) = n implies n > 0, but I didn't take off if you failed to say that explicitly.
- If s has the form skip; s_2, then s' is s_2 and inverting mysize(s) = n ensures 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 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.
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)
4. Consider a \( \lambda \)-calculus with pairs built-in. That is, \((v_1, v_2)\) is a value if \(v_1\) and \(v_2\) are values, \((v_1, v_2).1 \rightarrow v_1\) and \((v_1, v_2).2 \rightarrow v_2\).

(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 \textit{encoding}, we mean you may \textit{not} extend the syntax of the language.)

(b) In the simply-typed \( \lambda \)-calculus with pairs (and types of the form \( \tau_1 \ast \tau_2 \)), give \textit{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 \( \tau \) such that \( \cdot \vdash e : \tau \).)

\textbf{Solution:}

(a) \( \text{“make-triple”} = \lambda x. \lambda y. \lambda z. ((x, y), z) \)
   \( \text{“first”} = \lambda x. x.1.1 \)
   \( \text{“second”} = \lambda x. x.1.2 \)
   \( \text{“third”} = \lambda x. x.2 \)

(b) \( \text{int->int->int->((int*int)*int)} \)
   \( \text{int->int->(int*int)->((int*int)*(int*int))} \)
   \( \ldots \)
5. Under what assumptions do the following terms type-check in the simply-typed $\lambda$-calculus? That is, for the given $e$, describe all $\Gamma$ and $\tau$ 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 $\Gamma$ and $\tau$ where $\Gamma$ maps $x$ to a type of the form $\tau_1 \rightarrow \tau$ and $y$ to a type of the form $\tau_1$.
(b) Any $\Gamma$ and $\tau$ where $\Gamma$ maps $f$ to a type of the form $\tau_1 \rightarrow \tau_1$ and $\tau = \tau_1 \rightarrow \tau_1$.
(c) Any $\Gamma$ and $\tau$ where $\tau$ has the form $\tau_1 \rightarrow \tau_2 \rightarrow \tau_1$.
(d) There is no $\Gamma$ and $\tau$ for which this program type-checks.
6. Recall how we extend the simply-typed λ-calculus with fix:

\[
\begin{align*}
  e &\rightarrow e' \\
  \text{fix } e &\rightarrow \text{fix } e' \\
  \text{fix } \lambda x. e &\rightarrow e[(\text{fix } \lambda x. e)/x] \\
  \Gamma \vdash e : \tau \rightarrow \tau &\quad \Gamma \vdash \text{fix } e : \tau
\end{align*}
\]

Also we recall that this extension is type-safe.

(a) If we add the rule

\[\Gamma \vdash \text{fix } e : \tau\]

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

\[\Gamma \vdash \text{fix } \lambda x. x : \tau\]

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 \(e \rightarrow e'\), 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 \(\text{fix } e\). So \(\text{fix } (3 \ 4)\) would get stuck.

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