CSE505, Winter 2012, Midterm Examination
February 7, 2012

Rules:

- The exam is closed-book, closed-notes, except for one side of one 8.5x11in piece of paper.
- **Please stop promptly at 10:20.**
- You can rip apart the pages, but please write your name on each page.
- There are **100 points** total, distributed **unevenly** among **5** 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.
For your reference:

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

\[
\begin{array}{c|c|c|c|c}
\text{CONST} & \text{VAR} & \text{ADD} & \text{MULT} \\
\hline
H; c \Downarrow c & H; x \Downarrow H(x) & H; e_1 \Downarrow c_1, H; e_2 \Downarrow c_2 & H; e_1 \ast e_2 \Downarrow c_1 \ast c_2 \\
\end{array}
\]

\[
\begin{align*}
H_1; s_1 & \rightarrow H_2; s_2 \\
\end{align*}
\]

\[
\begin{align*}
\text{ASSIGN} & & \text{SEQ1} & & \text{SEQ2} \\
H; e \Downarrow c & & H; \text{skip}; s \rightarrow H ; s & & H; s_1 \rightarrow H' ; s'_1 \\
\end{align*}
\]

\[
\begin{align*}
\text{IF1} & \quad H; e \Downarrow c & & \text{IF2} & \quad H; e \Downarrow c \\
H; \text{if } e \ s_1 & \rightarrow H ; s_1 & & H; \text{if } e \ s_1 & \rightarrow H ; s_2 \\
\end{align*}
\]

\[
\begin{align*}
\text{WHILE} & \quad H; \text{while } e \ s \rightarrow H ; \text{if } e \ (s; \text{while } e \ s) \text{ skip} \\
\end{align*}
\]

\[
\begin{align*}
\quad e & ::= \lambda x. e \mid x \mid e \mid c \\
\quad v & ::= \lambda x. e \mid c \\
\quad \tau & ::= \text{int} \mid \tau \rightarrow \tau \\
\end{align*}
\]

\[
\begin{align*}
\quad e \rightarrow e' \\
\quad \langle \lambda x. e \rangle v \rightarrow e[v/x] \\
\quad e_1 \rightarrow e'_1 \\
\quad e_1 \ e_2 \rightarrow e'_1 \ e_2 \\
\quad e_2 \rightarrow e'_2 \\
\quad v \ e_2 \rightarrow v \ e'_2 \\
\end{align*}
\]

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

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

- Preservation: If \( \cdot \vdash e : \tau \) and \( e \rightarrow e' \), then \( \cdot \vdash e' : \tau \).
- Progress: If \( \cdot \vdash e : \tau \), then \( e \) is a value or there exists an \( e' \) such that \( e \rightarrow e' \).
- Substitution: If \( \Gamma, x : \tau' \vdash e : \tau \) and \( \Gamma \vdash e' : \tau' \), then \( \Gamma \vdash e[e'/x] : \tau \).
1. (35 points) 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.

\[
\begin{align*}
  e & ::= \ldots \mid \text{read} \\
  s & ::= \ldots \mid \text{toggle} \\
  t & ::= \text{up} \mid \text{down}
\end{align*}
\]

(a) Give all the inference rules for large-step expression evaluation.

(b) Give all the inference rules for small-step statement evaluation.

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

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

Solution:

(a)

\[
\begin{align*}
  & H ; t ; e \Downarrow c \\
  \quad & H ; t \quad H(x) \\
  \quad & H ; t ; e_1 \Downarrow c_1, H ; t ; e_2 \Downarrow c_2 \\
  \quad & H ; t ; e_1 \ast e_2 \Downarrow c_1 \ast c_2 \\
  \quad & H ; t ; e_1 \Downarrow c_1 + e_2 \Downarrow c_1 + c_2 \\
  \quad & H ; t ; e_2 \Downarrow c_2 \\
  \quad & H ; t ; e \Downarrow c \quad H \uparrow ; \text{read} \Downarrow 1 \\
  \quad & H \downarrow ; \text{read} \Downarrow 0 \\
\end{align*}
\]

(b)

\[
\begin{align*}
  & H ; t ; e \Downarrow c \\
  \quad & H ; t ; x := e \rightarrow H, x \rightarrow c ; t; \text{skip} \\
  \quad & H ; t ; s_1 \rightarrow H' ; t' ; s' \quad H ; t ; s_1 ; s_2 \rightarrow H' ; t' ; s' ; s_2 \\
  \quad & H ; t ; \text{if } e_1 s_1 \rightarrow H ; t ; s_1 \\
  \quad & H ; t ; \text{if } e_1 s_2 \rightarrow H ; t ; s_2 \\
  \quad & H ; t ; \text{while } e s \rightarrow H ; t ; \text{if } e \text{ (s; while e s) \text{ skip}} \\
  \quad & H ; \text{up} \uparrow ; \text{toggle} \rightarrow H ; \text{down} ; \text{skip} \\
  \quad & H \downarrow ; \text{toggle} \rightarrow H ; \text{up} ; \text{skip}
\end{align*}
\]

(c) see next page

(d) see next page
(c) This statement is true. We prove it by induction on the derivation of \( H \uparrow e \vdash c \), 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 \( e = e_1 + e_2 \) for some \( e_1 \) and \( e_2 \), then \( H \uparrow e_1 \vdash c_1 \) and \( H \uparrow e_2 \vdash c_2 \) where \( c = c_1 + c_2 \). So by induction \( H \uparrow e_1' \vdash c_1' \) and \( H \uparrow e_2' \vdash c_2' \) where \( e_1' \) and \( e_2' \) are \( e_1 \) and \( e_2 \) with \texttt{read} replaced by 1. So we can use the rule for addition to derive \( H \uparrow e_1' + e_2' \vdash c_1' + c_2' \). This is what we need because \( e_1' + e_2' = e \) with \texttt{read} replaced by 1 and \( c = c_1 + c_2 \).
- If \( e = e_1 \ast e_2 \) for some \( e_1 \) and \( e_2 \), then \( H \uparrow e_1 \vdash c_1 \) and \( H \uparrow e_2 \vdash c_2 \) where \( c = c_1 \ast c_2 \). So by induction \( H \uparrow e_1' \vdash c_1' \) and \( H \uparrow e_2' \vdash c_2' \) where \( e_1' \) and \( e_2' \) are \( e_1 \) and \( e_2 \) with \texttt{read} replaced by 1. So we can use the rule for multiplication to derive \( H \uparrow e_1' \ast e_2' \vdash c_1' \ast c_2' \). This is what we need because \( e_1' \ast e_2' = e \) with \texttt{read} replaced by 1 and \( c = c_1 \ast c_2 \).
- If \( e \) is \texttt{read} and the toggle is \texttt{up}, then \( c = 1 \) and \( e' = 1 \) and we can use the rule for constants to derive \( H \uparrow e \vdash 1 \).
- The rule where \( e \) is \texttt{read} and the toggle is \texttt{down} cannot end the derivation of \( H \uparrow e \vdash c \), so this case holds vacuously.

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

\[
\begin{align*}
. &; \text{up; toggle; (} x := \text{read; toggle} \rightarrow^* ., x \mapsto 0 ; \text{up; skip}, \\
\text{but} &;
. &; \text{up; toggle; (} x := 1; \text{toggle} \rightarrow^* ., x \mapsto 1 ; \text{up; skip}
\end{align*}
\]
2. (31 points) This problem uses Caml and continues using IMP-with-toggle from problem 1. You are given the type definitions for IMP-with-toggle and the "mysterious" function foo:

```caml
let foo lst =
  let rec f lst s =
    match lst with
    [] -> true
    | hd::tl -> hd <> s && f tl s in (* <> is "not equal" *)
  let rec g i =
    let t = "_t" ^ (string_of_int i) in (* ^ concatenates strings *)
    if f lst t then t else g (i+1) in
  g 0
```

(a) Document foo: What does it take and what does it return (in terms of types and values)? Do not describe how foo is implemented.

(b) Write a Caml function allVars of type stmt -> string list that returns all the variables appearing anywhere in the statement. Hints:
   • Duplicate strings are fine; do not bother removing them.
   • Sample solution is approximately 15 lines total.
   • You will need a helper function.
   • Caml's append operator @ is very useful.

(c) IMP-with-toggle is kind of stupid because we can encode the concept in regular IMP. Describe in 1–3 English sentences how you could translate IMP-with-toggle to regular IMP.

(d) Implement the translation you described in part (c) with a Caml function translate of type stmt -> stmt. Hints:
   • The result should not use Toggle or Read.
   • The sample solution is approximately 20 lines total.
   • You will need a helper function.
   • There's a reason parts (a) and (b) are part of this problem.

Solution:

(next page)
Solution:

(a) `foo` has type `string list -> string`. It returns the string `_ti` where `i` is (the string representation of) the smallest natural number such that `_ti` is not in the argument list.

(b) let rec allVarsE e =
    match e with
    | Int _ -> []
    | Var s -> [s]
    | Plus(e1,e2) -> (allVarsE e1) @ (allVarsE e2)
    | Times(e1,e2) -> (allVarsE e1) @ (allVarsE e2)
    | Read -> []

let rec allVars s =
    match s with
    | Skip -> []
    | Toggle -> []
    | Assign(x,e) -> x :: (allVarsE e)
    | If(e,s1,s2) -> (allVarsE e) @ (allVars s1) @ (allVars s2)
    | While(e,s1) -> (allVarsE e) @ (allVars s1)
    | Seq(s1,s2) -> (allVars s1) @ (allVars s2)

(c) We can use a new IMP variable `x` not otherwise used in the program to hold the current toggle (1 for `up`, 0 for `down`). Then we can replace `read` with reading `x` and `toggle` with `x := 1 + (-1 * x)` (or if `x := 0` then `x := 1`).

(d) let translate s =
    let v = foo (allVars s) in
    let rec xe e =
        match e with
        | Int _ -> e
        | Var _ -> e
        | Plus(e1,e2) -> Plus(xe e1, xe e2)
        | Times(e1,e2) -> Times(xe e1, xe e2)
        | Read -> Var v in
    let rec xs s =
        match s with
        | Skip -> Skip
        | Toggle -> Assign(v,Plus(Int 1,Times(Int (-1),Var v)))
            (* alternately If(Var(v),Assign(v,Int 0),Assign(v,Int 1)) *)
        | Assign(x,e) -> Assign(x, xe e)
        | If(e,s1,s2) -> If(xe e, xs s1, xs s2)
        | Seq(s1,s2) -> Seq(xs s1, xs s2)
        | While(e,s) -> While(xe e, xs s)
    in xs s
3. (13 points) This problem uses the untyped lambda-calculus and full reduction. Recall this encoding of pairs:

- “mkpair” \( \lambda x. \lambda y. \lambda z. z \times y \)
- “fst” \( \lambda p. p \times x. \lambda y. x \)
- “snd” \( \lambda p. p \times x. \lambda y. y \)

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”:

\[
\begin{align*}
(\lambda p. p \times x. \lambda y. x)(\lambda x. \lambda y. \lambda z. z \times y)(z \times z) & \\
& \rightarrow (\lambda p. p \times x. \lambda y. x)(\lambda y. \lambda z. z \times y)(z) \\
& \rightarrow (\lambda p. p \times x. \lambda y. x)(\lambda z. z \times (z \times z)) \\
& \rightarrow (\lambda z. z \times (z \times z)) \times x. \lambda y. x \\
& \rightarrow (\lambda x. \lambda y. x)(\lambda x. \lambda y. x)(\lambda x. \lambda y. x) \\
& \rightarrow \lambda x. \lambda y. x \\
& \rightarrow \lambda x. \lambda y. x
\end{align*}
\]

(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 \( \lambda z. z \times x \) in order to perform these first two steps properly.

(b)

\[
\begin{align*}
(\lambda p. p \times x. \lambda y. x)(\lambda x. \lambda y. \lambda z. z \times y)(z \times z) & \\
& \rightarrow (\lambda p. p \times x. \lambda y. x)(\lambda y. \lambda q. q \times y)(z) \\
& \rightarrow (\lambda p. p \times x. \lambda y. x)(\lambda q. q \times (z \times z)) \\
& \rightarrow (\lambda q. q \times (z \times z)) \times x. \lambda y. x \\
& \rightarrow (\lambda x. \lambda y. x)(z \times z) \\
& \rightarrow (\lambda y. z)(z) \\
& \rightarrow z
\end{align*}
\]
4. (10 points) In this problem, assume the simply-typed lambda calculus. For each of the following:

- If the answer is yes, give an example Γ and τ.
- If the answer is no, you can just say “no.”

(a) Is there a Γ and τ such that Γ ⊢ (λx. x) x : τ ?
(b) Is there a Γ and τ such that Γ ⊢ λx. (x x) : τ ?
(c) Is there a Γ and τ such that Γ ⊢ x x : τ ?
(d) Is there a Γ and τ such that Γ ⊢ x (λx. x) : τ ?

Solution:

(a) Yes, for example Γ = ·, x: int and τ = int. In general, the type of x in Γ has to be τ.
(b) No
(c) No
(d) Yes, for example Γ = ·, x:(int → int) → int and τ = int. In general, the type of x in Γ has to have the form (τ’ → τ’) → τ.
5. (11 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.
(d) Explain why we proved the Preservation Lemma instead of just the Differently Preserved Lemma.

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.
(d) Because the Differently Preserved Lemma has too weak an induction hypothesis for the proof to go through. For example, when $e_1 e_2 \rightarrow e'_1 e_2$ because $e_1 e'_1$, it’s not enough to know that $e'_1$ has some type. We need to know it has the same type as $e_1$ to show that $e'_1 e_2$ still type-checks.