Please do not turn the page until everyone is ready.

Rules:

• The exam is closed-book, closed-note, except for one side of one 8.5x11in piece of paper.

• Please stop promptly at 12:20.

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

• There are 120 points total, distributed evenly among 6 questions (most of 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.

• If you have questions, ask.

• Relax. You are here to learn.
e → e' and Γ ⊢ e : τ and τ₁ ≤ τ₂

\[
\begin{align*}
\frac{e_1 → e'_1 \quad e_2 → e'_2 \quad e → e'}{v e_1 e_2 → v e'_1 e'_2} & \quad \frac{\text{fix } e → \text{fix } e'}{\text{fix } \lambda x. e → \text{fix } \lambda x. e / x}\end{align*}
\]

\[
\frac{\{l_1 = v_1, \ldots, l_n = v_n\} \downarrow \rightarrow v_i}{e_i → e'_i}
\]

\[
\frac{\{l_1 = v_1, \ldots, l_{i-1} = v_{i-1}, l_i = e_i, \ldots, l_n = e_n\} \rightarrow \{l_1 = v_1, \ldots, l_{i-1} = v_{i-1}, l_i = e'_i, \ldots, l_n = e_n\}}{\Gamma \vdash e : \tau → \tau}
\]

\[
\frac{\Gamma \vdash \{l_1 = e_1, \ldots, l_n = e_n\} : \{l_1 : \tau_1, \ldots, l_n : \tau_n\}}{\Gamma \vdash \{l_1 : \tau_1', \ldots, l_n : \tau_n, l : \tau\} \leq \{l_1 : \tau_1, \ldots, l_n : \tau_n\}}
\]

\[
\frac{\tau_1 \leq \tau'_1 \quad \tau_2 \leq \tau_3 \quad \tau_4 = \tau}{\{l_1 : \tau_1, \ldots, l_n : \tau_n\} \leq \{l_1, \ldots, l_n : \tau\} \leq \{l_1 : \tau_1', \ldots, l_n : \tau_n\}}
\]

\[
\frac{e \iff c \mid \lambda x : \tau. e \mid e e \mid \Lambda \alpha. e \mid e[\tau]}{\tau \iff \text{int} \mid \tau \rightarrow \tau \mid \alpha \mid \forall \alpha. \tau}
\]

e → e' and Δ; Γ ⊢ e : τ

\[
\begin{align*}
\frac{e \rightarrow e' \quad e e_2 \rightarrow e' e_2}{v e \rightarrow v e' \quad e[\tau] \rightarrow e'[\tau]} & \quad \frac{\lambda x : \tau. e v \rightarrow e[v / x]}{\Lambda \alpha. e[\tau] \rightarrow e[\tau / \alpha]}\end{align*}
\]

\[
\frac{\Delta; \Gamma \vdash x : \Gamma(x)}{\Delta; \Gamma \vdash c : \text{int}} & \quad \frac{\Delta; \Gamma \vdash \lambda x : \tau_1. e : \tau_1 \rightarrow \tau_2}{\Delta; \Gamma \vdash \lambda \alpha. e : \forall \alpha. \tau_1 \rightarrow \tau_2}
\]

\[
\frac{\Delta; \Gamma \vdash e_1 : \tau_2 \rightarrow \tau_1 \quad \Delta; \Gamma \vdash e_2 : \tau_2}{\Delta; \Gamma \vdash e_1 e_2 : \tau_1} & \quad \frac{\Delta; \Gamma \vdash \forall \alpha. \tau_1 \rightarrow \tau_2}{\Delta; \Gamma \vdash e[\tau] \rightarrow e[\tau / \alpha]}\end{align*}
\]

\[
\frac{\Delta; \Gamma \vdash e_1 : \tau_2 \rightarrow \tau_1 \quad \Delta; \Gamma \vdash e_2 : \tau_2}{\Delta; \Gamma \vdash \lambda \alpha. e : \forall \alpha. \tau_1 \rightarrow \tau_2}
\]
\[ e ::= c \mid \lambda x.\ e \mid e\ e \mid (e, e) \mid e.1 \mid e.2 \mid \text{letcc}\ x.\ e \mid \text{throw}\ e \mid \text{continuation}\ E \]
\[ E ::= [\!] \mid E \circ E \mid (E, E) \mid (v, E) \mid E.1 \mid E.2 \mid \text{throw}\ E \mid \text{throw}\ v \ E \]
\[ v ::= c \mid \lambda x.\ e \mid (v, v) \mid \text{continuation}\ E \]
\[ \frac{e \vdash e'}{E[e] \rightarrow E[e']} \quad \frac{E[\text{letcc}\ x.\ e] \rightarrow E[e[\text{continuation}\ E/x]]}{E[\text{throw}(\text{continuation}\ E')\ v] \rightarrow E'[v]} \]
\[ e ::= \ldots \mid \text{inl}(e) \mid \text{inr}(e) \mid (\text{case}\ e\ .\ e \mid x.e) \mid \text{roll}_r\ e \mid \text{unroll}\ e \mid \text{try}\ e \mid \text{catch}(c)\ e \]
\[ \tau ::= \ldots \mid \tau_1 \tau_2 \mid \mu\alpha\tau \]
\[ v ::= \ldots \mid \text{inl}(v) \mid \text{inr}(v) \mid \text{roll}_v \]
\[ \text{case}\ \text{inl}(v)\ x.e_1 \ |\ x.e_2 \rightarrow e_1[v/x] \quad \text{case}\ \text{inr}(v)\ x.e_1 \ |\ x.e_2 \rightarrow e_2[v/x] \quad \text{inl}(e) \rightarrow \text{inl}(e') \]
\[ e \rightarrow e' \quad \frac{\text{inr}(e) \rightarrow \text{inr}(e')}{e \rightarrow e'} \quad \frac{\text{case}\ e\ .\ e_1 \ |\ x.e_2 \rightarrow \text{case}\ e'\ .\ e_1 \ |\ x.e_2}{e \rightarrow e'} \quad \frac{\text{roll}_{\mu\alpha\tau}\ e \rightarrow \text{roll}_{\mu\alpha\tau}\ e'}{e \rightarrow e'} \]
\[ \text{unroll}\ e \rightarrow \text{unroll}\ e' \quad \frac{\text{unroll}(\text{roll}_{\mu\alpha\tau}\ v) \rightarrow v}{e \rightarrow e'} \quad \frac{\text{raise}\ e \rightarrow \text{raise}\ e'}{e \rightarrow e'} \]
\[ \text{try}\ e_1\ \text{catch}(c)\ e_2 \rightarrow \text{try}\ e_1'\ \text{catch}(c)\ e_2 \quad \text{try}\ v\ \text{catch}(c)\ e_2 \rightarrow v \quad \text{try}\ \text{raise}\ c\ \text{catch}(c)\ e_2 \rightarrow e_2 \]
\[ c \neq e' \quad \frac{\text{try}\ \text{raise}\ c'\ \text{catch}(c)\ e_2 \rightarrow \text{raise}\ c'}{\text{many “bubble up exception” rules omitted}} \]
\[ \Delta;\Gamma \vdash e : \tau_1 \quad \frac{\Delta;\Gamma \vdash e : \tau_2}{\Delta;\Gamma \vdash \text{inl}(e) : \tau_1 \tau_2} \quad \frac{\Delta;\Gamma \vdash \text{inr}(e) : \tau_1 + \tau_2}{\Delta;\Gamma \vdash \text{case}\ e\ .\ e_1 \ |\ x.e_2 : \tau} \]
\[ \frac{\Delta;\Gamma \vdash e : \tau[(\mu\alpha\tau)/\alpha]}{\Delta;\Gamma \vdash e : \mu\alpha\tau} \quad \frac{\Delta;\Gamma \vdash \text{roll}_{\mu\alpha\tau}\ e : \mu\alpha\tau}{\Delta;\Gamma \vdash \text{unroll}\ e : \tau[(\mu\alpha\tau)/\alpha]} \quad \frac{\Delta;\Gamma \vdash e_1 : \tau}{\Delta;\Gamma \vdash \text{try}\ e_1\ \text{catch}(c) : e_2 : \tau} \]
\[ \frac{e ::= \ldots \mid \text{ref}\ e \mid \text{let}\ e_1 := e_2 \mid r}{(H; e) \rightarrow (H'; e')} \quad \frac{v ::= \ldots \mid \text{ref}}{v \not\in \text{Dom}(H)} \quad \frac{H; e_1 \rightarrow H'; e_1'}{H ; e_1 \rightarrow (H' ; e_1')} \quad \frac{r \not\in \text{Dom}(H)}{r \not\in \text{Dom}(H)} \quad \frac{H ; r := v \rightarrow (H ; r := v ; v)}{(H ; r := v) \rightarrow (H ; r := v ; v)} \quad \frac{(H ; r := v) \rightarrow (H ; r := v ; v)}{(H ; r := v) \rightarrow (H ; r := v ; v)} \]
\[ \frac{\Delta;\Gamma \vdash e : \tau}{\Delta;\Gamma \vdash \text{ref}\ e : \tau \text{ ref}} \quad \frac{\Delta;\Gamma \vdash e : \tau \text{ ref}}{\Delta;\Gamma \vdash \text{un}\ e : \tau} \quad \frac{\Delta;\Gamma \vdash e_1 : \tau \text{ ref}}{\Delta;\Gamma \vdash \text{ref}\ e_1 \rightarrow \text{ref}\ e_2 : \tau} \quad \frac{\Delta;\Gamma \vdash e_1 : \tau \text{ ref}}{\Delta;\Gamma \vdash \text{ref}\ e_1 := e_2 : \text{unit}} \]
\[ \text{many inductive rules omitted} \]
1. Here are two type definitions for different representations of linked lists of integers:

- type \( t_1 = \mu \alpha. (\text{unit} + (\text{int} \times \alpha)) \)
- type \( t_2 = \mu \alpha. ((\alpha \times \text{int}) + \text{unit}) \)

Write a typed \( \lambda \)-calculus program of the form \( \text{fix}(\lambda \text{convert} : \_. \lambda \text{lst} : \_. \_. \_.) \) for converting a list of type \( t_1 \) to a list of type \( t_2 \).

Your program should typecheck \textit{without} subtyping (i.e., you should use \texttt{roll} and \texttt{unroll} along with \texttt{case}, pair operations, etc.).

You may use \( t_1 \) and \( t_2 \) as abbreviations for their definitions if you wish.

\textbf{20 points}
2. Consider the following proposed changes to System F separately and explain why each is a bad idea.

(a) Replace the typing rule on the left with the typing rule on the right:

\[
\frac{\Delta, \alpha; \Gamma \vdash e : \tau_1}{\Delta; \Gamma \vdash \Lambda \alpha. e : \forall \alpha.\tau_1} \quad \frac{\Delta; \Gamma \vdash e[\tau_2/\alpha] : \tau_1}{\Delta; \Gamma \vdash \Lambda \alpha. e : \forall \alpha.\tau_1}
\]

(b) Replace the typing rule on the left with the typing rule on the right:

\[
\frac{\Delta; \Gamma \vdash e : \forall \alpha.\tau_1 \quad \Delta \vdash \tau_2}{\Delta; \Gamma \vdash e[\tau_2] : \tau_1[\tau_2/\alpha]} \quad \frac{\Delta; \Gamma \vdash e : \forall \alpha.\tau_1 \quad \Delta \vdash \tau_2}{\Delta; \Gamma \vdash e[\tau_2] : \forall \alpha.\tau_1}
\]

10 points each
3. In this problem, assume Caml has letcc and throw in addition to (and separate from) try and raise.

Consider the following programs separately. For each:

• What does it print?
• What is the type of \( f \)?

Partial credit will require explanation of your answers.

Part (d) is difficult.

5 points each

(a) exception Foo
   let f () = (print_string "A"; raise Foo)
   let x = try f() with Foo -> f()

(b) exception Foo
   let f () = (print_string "A"; Foo)
   let x = try f() with Foo -> f()

(c) let f () =
   let rec g i k =
       if i > 0
       then (print_string "A"; g (i-1) k; print_string "B"; 7)
       else throw k 7
   in
   (letcc k. g 3 k)
   let x = f()

(d) let f () =
   let k = ref None
   let rec g i =
       if i > 0
       then (print_string "A"; g (i-1); print_string "B"; 7)
       else (letcc k2. ((k := Some k2); 7))
   in
   (g 3;
    match !k with None -> 7 | Some k2 -> (k := None; throw k2 7))
   let x = f()
4. Java interfaces do not allow fields, but suppose they did.

For each rule below, determine if it is sound or unsound. If it is unsound, give a short (around 10 lines, including class and interface definitions) example program that would typecheck but get stuck at run-time. Do not worry about syntax, making a correct main method, etc.

Recall that if interface $I$ extends interface $J$, then $I \leq J$.
Also recall that a `final` field can be read but not written.

Assume interface $J$ has a field $f$ of type $T$.

(a) If $f$ is non-final, interface $I$ may extend interface $J$ by changing $f$ to have a subtype of $T$.
(b) If $f$ is non-final, interface $I$ may extend interface $J$ by changing $f$ to have a supertype of $T$.
(c) If $f$ is final, interface $I$ may extend interface $J$ by changing $f$ to have a subtype of $T$.
(d) If $f$ is final, interface $I$ may extend interface $J$ by changing $f$ to have a supertype of $T$.

20 points total, graded together

---

1Technically, Java allows `public static final` fields, but this problem considers instance fields.
5. Suppose we *change the semantics* of Java so that method-lookup uses multimethods instead of static overloading.

True or false. **Briefly explain your answers.**

(a) If all methods in program $P$ take 0 arguments (that is, all calls look like $e.m()$), then $P$ definitely behaves the same after the change.

(b) If all methods in program $P$ take 1 argument (that is, all calls look like $e.m(e')$), then $P$ definitely behaves the same after the change.

(c) If a program $P$ typechecks without ever using subsumption, then $P$ definitely behaves the same after the change.

(d) Given an arbitrary program $P$, it is decidable whether $P$ behaves the same after the change.

5 points each
6. In the simply-typed λ-calculus with records and without subtyping, the following is true by inspection of the typing rules:

If \( \cdot \vdash v : \{l_1 : \tau_1, l_2 : \tau_2\} \), then there exist \( v_1 \) and \( v_2 \) such that \( v = \{l_1 = v_1, l_2 = v_2\} \).

(a) Explain why the statement above is false in the presence of subtyping. In particular, which subtyping rules make it false?

6 points

(b) Revise the claim so that it is true but as strong as possible. That is, complete this sentence, “If \( \cdot \vdash v : \{l_1 : \tau_1, l_2 : \tau_2\} \), then \( v \) is ...” with a fact that requires the assumed typing derivation. You can state the claim in English but be precise.

6 points

(c) Prove your revised claim. (Hints: Use a “helper” lemma about subtyping derivations where the supertype is a record type containing certain fields. You will need induction and a strengthened induction hypothesis to prove the claim in part (b); it turns out the helper lemma does not need induction.)

8 points
This page is intentionally blank.