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.
- There are 100 points total, distributed unevenly among 7 questions.
- The questions 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.
Name: __________________________

For your reference (page 1 of 2):

\[
\begin{align*}
\Gamma, x: \tau & \vdash e: \tau_2 \\
\Gamma & \vdash e_1: \tau_2 \\
\Gamma & \vdash e_2: \tau_2 \\
\Gamma & \vdash e : \tau_2 \\
\Gamma & \vdash \fix e: \tau
\end{align*}
\]

\[
\begin{align*}
\{l_1 = v_1, \ldots, l_n = v_n\} & \vdash v \rightarrow v_1 \\
\{l_1 = v_1, \ldots, l_{i-1} = v_{i-1}, l_i = e_i, \ldots, l_n = v_n\} & \vdash \{l_1 = v_1, \ldots, l_{i-1} = v_{i-1}, l_i = e'_i, \ldots, l_n = v_n\}
\end{align*}
\]

\[
\begin{align*}
\Gamma & \vdash c: \text{int} \\
\Gamma & \vdash x: \Gamma(x) \\
\Gamma & \vdash \lambda x. e: \tau_1 \rightarrow \tau_2 \\
\Gamma & \vdash e_1: \tau_2 \\
\Gamma & \vdash e_2: \tau_2 \\
\Gamma & \vdash \fix e: \tau
\end{align*}
\]

\[
\begin{align*}
\{l_1 = e_1, \ldots, l_n = e_n\} & \vdash \{l_1: \tau_1, \ldots, l_n: \tau_n\}
\end{align*}
\]

\[
\begin{align*}
\{l_1: \tau_1, \ldots, l_n: \tau_n, \tau\} & \leq \{l_1: \tau_1, \ldots, l_n: \tau_n\}
\end{align*}
\]

\[
\begin{align*}
\{l_1, \tau_1, \ldots, l_{i-1}, \tau_{i-1}, l_i, \tau_i, \ldots, l_n: \tau_n\} & \leq \{l_1, \tau_1, \ldots, l_i, \tau_i, l_{i+1}, \tau_{i+1}, \ldots, l_n: \tau_n\}
\end{align*}
\]

\[
\begin{align*}
t_i & \leq \tau_i
\end{align*}
\]

\[
\begin{align*}
t_3 & \leq \tau_1 \\
t_2 & \leq \tau_2 \\
t_1 & \rightarrow \tau_3 \rightarrow \tau_4 \\
\tau & \leq \tau \\
\tau_3 & \leq \tau_2 \\
\tau_2 & \leq \tau_3 \\
\tau_1 & \rightarrow \tau_4 \\
\tau & \leq \tau_3 \\
\tau_1 & \leq \tau_2 \\
\tau_1 & \leq \tau_3
\end{align*}
\]

\[
\begin{align*}
\Gamma & ::= \cdot | \Gamma, x: \tau \\
\Delta & ::= \cdot | \Delta, \alpha
\end{align*}
\]

\[
\begin{align*}
\Gamma \vdash e : \tau \\
\Delta, \Gamma \vdash x: \Gamma(x) \\
\Delta, \Gamma \vdash c: \text{int} \\
\Delta, \Gamma \vdash \lambda x. \alpha. e: \tau_1 \rightarrow \tau_2 \\
\Delta, \Gamma \vdash e_1: \tau_2 \\
\Delta, \Gamma \vdash e_2: \tau_2 \\
\Delta, \Gamma \vdash \forall \alpha. \tau_1 \\
\Delta, \Gamma \vdash e[\tau_2: \tau_1/\alpha]
\end{align*}
\]
\[
\begin{align*}
e &::= \ldots | A(e) | B(e) | (\text{match } e \text{ with } Ax. e | Bx. e) | \text{roll}_\tau e | \text{unroll} e | (e, e) | e.1 | e.2 \\
\tau &::= \ldots | \tau_1 + \tau_2 | \mu_\alpha \tau | \tau_1 * \tau_2 \\
v &::= \ldots | A(v) | B(v) | \text{roll}_\tau v | (v, v)
\end{align*}
\]

\[e \rightarrow e' \text{ and } \Delta; \Gamma \vdash e : \tau\]

\[
\begin{array}{c}
A(e) \rightarrow A(e') & B(e) \rightarrow B(e') & \text{match } e \text{ with } Ax. e_1 | By. e_2 \rightarrow \text{match } e' \text{ with } Ax. e_1 | By. e_2 \\
\end{array}
\]

\[
\begin{array}{c}
\text{match } A(v) \text{ with } Ax. e_1 | By. e_2 \rightarrow e_1[v/x] & \text{match } B(v) \text{ with } Ax. e_1 | By. e_2 \rightarrow e_2[v/y] \\
\end{array}
\]

\[
\begin{array}{c}
\text{roll}_\mu_\alpha \tau e \rightarrow \text{roll}_\mu_\alpha \tau e' & \text{unroll } e \rightarrow \text{unroll } e' & \text{unroll } (\text{roll}_\mu_\alpha \tau v) \rightarrow v \\
\end{array}
\]

\[
\begin{array}{c}
e_1 \rightarrow e_1' & e_2 \rightarrow e_2' & (v, e_2) \rightarrow (v, e_2') & e.1 \rightarrow e'.1 & e.2 \rightarrow e'.2 & (v, v_1).1 \rightarrow v_1 & (v, v_2).2 \rightarrow v_2 \\
\end{array}
\]

\[
\Delta; \Gamma \vdash e : \tau_1 + \tau_2 & \Delta; \Gamma \vdash e_1 : \tau & \Delta; \Gamma \vdash e_2 : \tau & \Delta; \Gamma \vdash \text{match } e \text{ with } Ax. e_1 | By. e_2 : \tau \\
\Delta; \Gamma \vdash A(e) : \tau_1 + \tau_2 & \Delta; \Gamma \vdash B(e) : \tau_1 + \tau_2 & \Delta; \Gamma \vdash e : \tau[(\mu_\alpha \tau)/\alpha] & \Delta; \Gamma \vdash \text{roll}_\mu_\alpha \tau e : \mu_\alpha \tau & \Delta; \Gamma \vdash \text{unroll } e : \tau[(\mu_\alpha \tau)/\alpha] \\
\Delta; \Gamma \vdash e_1 : \tau_1 & \Delta; \Gamma \vdash e_2 : \tau_2 & \Delta; \Gamma \vdash (e_1, e_2) : \tau_1 * \tau_2 & \Delta; \Gamma \vdash e : \tau_1 * \tau_2 & \Delta; \Gamma \vdash e.1 : \tau_1 & \Delta; \Gamma \vdash e.2 : \tau_2
\end{array}
\]

Module Thread:

\[
type t
\]
\[
val create : ('a -> 'b) -> 'a -> t
\]
\[
val join : t -> unit
\]

Module Mutex:

\[
type t
\]
\[
val create : unit -> t
\]
\[
val lock : t -> unit
\]
\[
val unlock : t -> unit
\]

Module Event:

\[
type 'a channel
\]
\[
type 'a event
\]
\[
val new_channel : unit -> 'a channel
\]
\[
val send : 'a channel -> 'a -> unit event
\]
\[
val receive : 'a channel -> 'a event
\]
\[
val choose : 'a event list -> 'a event
\]
\[
val wrap : 'a event -> ('a -> 'b) -> 'b event
\]
\[
val sync : 'a event -> 'a
\]
1. (15 points) Consider a typed-lambda calculus with functions, integers, records, and subtyping as considered in class. Note this problem considers only the subtyping judgment with the six inference rules on page 2 of this exam, not the typing judgment. For each of the following claims, if it is true, prove it. If it is false, provide a counterexample.

(a) If $\tau_1 \leq \tau_2$ and $\tau_1$ is a record type, then $\tau_2$ is a record type.

(b) If $\tau_1 \leq \tau_2$ and $\tau_1$ contains a function type somewhere in it, then $\tau_2$ contains a function type somewhere in it.

Solution:

(a) True. Proof by induction on the derivation of $\tau_1 \leq \tau_2$ proceeding by cases on the bottom-most rule instantiated in the derivation:

- If width subtyping, then $\tau_2$ is a record type.
- If permutation subtyping, then $\tau_2$ is a record type.
- If depth subtyping on records, then $\tau_2$ is a record type.
- If function subtyping, then $\tau_1$ is not a record type so the claim holds vacuously.
- If reflexivity, then $\tau_1 = \tau_2$, so $\tau_1$ being a record type implies $\tau_2$ is a record type.
- If transitivity, then by inversion there is some $\tau$ such that $\tau_1 \leq \tau$ and $\tau \leq \tau_2$. Assume $\tau_1$ is a record type. Then by induction and $\tau_1 \leq \tau$, $\tau$ is a record type. Then by induction and $\tau \leq \tau_2$, $\tau_2$ is a record type.

(b) False. One example: Let $\tau_1 = \{ l_1 : \tau_2 \to \tau_3 \}$ and $\tau_2 = \{ \}$. Width subtyping suffices.
2. (10 points) Consider System F.

(a) Write a well-typed term that implements function composition and is as polymorphic as possible. Function composition takes two (curried) arguments (say \( f \) and \( g \)) and returns a function that given \( x \) returns \( f(g(x)) \).

(b) Give the type for the term you wrote in part (a).

Be sure to use parentheses appropriately.

Solution:

(a) \( \Lambda \alpha_1. \Lambda \alpha_2. \Lambda \alpha_3. \lambda f : \alpha_2 \rightarrow \alpha_3. \lambda g : \alpha_1 \rightarrow \alpha_2. \lambda x : \alpha_1. f(g x) \)

(b) \( \forall \alpha_1. \forall \alpha_2. \forall \alpha_3. (\alpha_2 \rightarrow \alpha_3) \rightarrow (\alpha_1 \rightarrow \alpha_2) \rightarrow \alpha_1 \rightarrow \alpha_3 \)
3. (10 points) For each of the following Caml definitions, does it type-check in Caml? If so, what type does it have? If not, why not?

(a) let part_a = (fun g -> (fun x y -> x) (g 0) (g 17))
(b) let part_b = (fun g -> (fun x y -> x) (g 0) (g true))
(c) let part_c = (fun g -> (fun x y -> x) (g 0) (g (g 17)))

Solution:

(a) Type-checks: (int -> 'a) -> 'a
(b) Does not type-check: The type-inferencer will conclude that g must be a function takes an int and a function that takes a bool, and these cannot both hold.
(c) Type-checks: (int -> int) -> int
4. (15 points) Consider a typed \( \lambda \)-calculus with recursive types, sums, pairs, int, string, and unit. Assume the language uses explicit roll and unroll coercions (not subtyping) for recursive types.

(a) Give a recursive type for binary trees where interior nodes have no data and each leaf has either a string or an int.

(b) In this part, you can use \( t \) as an abbreviation for the type you gave in part (a). Using \( \text{fix} \) for recursion, write a program of type \( t \rightarrow (\text{int} + \text{unit}) \). When called with a tree, the program should return \( A(i) \) if \( i \) is the left-most integer in the tree and \( B(()) \) if the tree has no integers. Give explicit types to all function arguments. If you get confused by \( \text{fix} \), use \( \text{let rec} \) instead for significant partial credit.

Solution:
Answer to (b) depends on answer to (a).

(a) One possible answer: \( \mu \alpha.((\text{string} + \text{int}) + (\alpha \times \alpha)) \).

(b) \begin{verbatim}
  fix \f : t -> (int + unit) .
  \x : t .
  match unroll x with
    A y. -> (match y with
             A z. -> B (())
            | B z. -> A (z))
    | B y. -> (match f y.1 with
               A z. -> A (z)
              | B z. -> f y.2)
\end{verbatim}
5. (20 points) In this problem you will use CML to implement a server for “rock-paper-scissors”. Rock-paper-scissors is a game where normally there are two players who each pick “rock”, “paper”, or “scissors” and either one player wins or there is a tie. This Caml code defines the rules for this two-player game:

```caml
type play = Rock | Scissors | Paper

let pick_winner p1 p2 = (* useful helper function *)
  match (p1,p2) with
    | (Rock,Rock) -> Tie
    | (Scissors,Scissors) -> Tie
    | (Paper,Paper) -> Tie
    | (Rock,Scissors) -> Left
    | (Rock,Paper) -> Right
    | (Scissors,Paper) -> Left
    | (Scissors,Rock) -> Right
    | (Paper,Rock) -> Left
    | (Paper,Scissors) -> Right
```

You will implement the `new_game` function in this interface:

```caml
type game
type play = Rock | Scissors | Paper

let new_game () = (* for you *)
```
let new_game () =
  let c = new_channel() in
  let send_results winner loser =
    sync(send winner Win);
    sync(send loser Lose) in
  let rec loop wait_kind lst =
    let play, response_ch = sync (receive c) in
    match lst with
    | [] -> loop play [response_ch]
    | hd::tl ->
      match pick_winner play wait_kind with
      | Tie -> loop wait_kind (response_ch::lst)
      | Left -> send_results response_ch hd; loop wait_kind tl
      | Right -> send_results hd response_ch; loop wait_kind tl in
  ignore(Thread.create (loop Rock) []); c
6. (15 points) Consider a class-based OOP language like we did in class where method-name reuse means overriding. Consider this code skeleton:

class A { A m() { return self; } ... }
class B extends A { B m() { return super(); } ... }
class C extends B { A m() { return new A(); } ... }
class Main { void main() { ... } }

(a) Assuming all code not shown (i.e., the code in the ...) type-checks, there are two reasons the code above does not type-check. What are they?
(b) One of your two answers to part (a) cannot actually lead to a program getting stuck. Explain why not.
(c) The other of your two answers to part (a) can lead to a program getting stuck. Fill in the ... as necessary (you may not need to use all of them) such that all the code you add type-checks but running the main method would produce a “method not found” error.

Solution:

(a) First, the return type of B’s m method is B, but the type of the super call is A, which is not a subtype of B. Second, C’s m method has return type A, but it is overriding a method with return type B, and again A is not a subtype of B.
(b) The first one (B’s m method) cannot cause a problem. Although the static type of super() is A, in fact this method returns self, which will be a B for any instance of B.
(c) Add to class B a method void n(). Then make the body of main be something like

```java
B c = new C();
c.m().n();
```

The first line type-checks using subsumption. The second line type-checks because B’s m returns a B and instance of B have an n method. But at run-time, c.m() returns an A, which does not have an n method.
7. (15 points) Consider a single-inheritance class-based OOP language. Assume booleans are provided as primitives.

Assume method-name reuse means either static overloading or multimethods. For each part give a single answer that is correct under either assumption. This is not intended to make the problem harder.

(a) Write a program (class definitions and client code) such that:

- The program type-checks.
- The program evaluates to true.
- There is one method definition you can remove from the program (comment-out) such that the program still type-checks but the program now evaluates to false.

Clearly indicate which method should be commented out.

(b) Write a program (class definitions and client code) such that:

- The program type-checks.
- The program evaluates to true.
- There is one method definition you can remove from the program (comment-out) such that the program would now have a “no best match” error.

Clearly indicate which method should be commented out.

Solution:

(a) class A {}
class B extends A{}
class Main{
    bool m(A a) { return false; }
    bool m(B b) { return true; } // This one!
    bool main() {
        return self.m(new B());
    }
}

(b) class A {}
class B extends A{}
class Main{
    bool m(A a, B b) { return true; }
    bool m(B b, A a) { return true; }
    bool m(B b, B B) { return true; } // This one!
    bool main() {
        return self.m(new B(), new B());
    }
}