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 96 points total, distributed unevenly among 5 questions.
- Most questions have multiple parts and the last question is much shorter and worth fewer points.

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 (page 1 of 2):

\[
\begin{align*}
\text{Name:} & \\
\text{For your reference (page 1 of 2):} & \\
\end{align*}
\]
\[ e ::= \ldots \mid A(e) \mid B(e) \mid (\text{match } e \text{ with } A x. \; e \mid B x. \; e) \mid \text{roll}_r \; e \mid \text{unroll} \; e \]
\[ \tau ::= \ldots \mid \tau_1 + \tau_2 \mid \mu \alpha. \tau \]
\[ v ::= \ldots \mid A(v) \mid B(v) \mid \text{roll}_\tau \; v \]
\[
\text{match } A(v) \text{ with } A x. \; e_1 | B y. \; e_2 \rightarrow e_1[v/x] \quad \text{match } B(v) \text{ with } A x. \; e_1 | B y. \; e_2 \rightarrow e_2[v/y]
\]
\[
\begin{array}{l}
e \rightarrow e' \\
A(e) \rightarrow A(e') \\
B(e) \rightarrow B(e') \\
\text{match } e \text{ with } A x. \; e_1 | B y. \; e_2 \rightarrow \text{match } e' \text{ with } A x. \; e_1 | B y. \; e_2 \\
\end{array}
\]
\[
\begin{array}{l}
\text{unroll} \; (\text{roll}_{\mu \alpha. \tau} \; v) \rightarrow v \\
\text{roll}_{\mu \alpha. \tau} \; e \rightarrow \text{roll}_{\mu \alpha. \tau} \; e' \\
\text{unroll} \; e \rightarrow \text{unroll} \; e'
\end{array}
\]
\[
\begin{array}{l}
\Delta; \Gamma \vdash e : \tau_1 + \tau_2 \\
\Delta; \Gamma, x : \tau_1 \vdash e_1 : \tau \\
\Delta; \Gamma, y : \tau_2 \vdash e_2 : \tau \\
\Delta; \Gamma \vdash \text{match } e \text{ with } A x. \; e_1 | B y. \; e_2 : \tau
\end{array}
\]

Module Thread:

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

Module Mutex:

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

Module Event:

\[
\text{type 'a channel}
\]
\[
\text{type 'a event}
\]
\[
\text{val new_channel : unit -> 'a channel}
\]
\[
\text{val send : 'a channel -> 'a -> unit event}
\]
\[
\text{val receive : 'a channel -> 'a event}
\]
\[
\text{val choose : 'a event list -> 'a event}
\]
\[
\text{val wrap : 'a event -> ('a -> 'b) -> 'b event}
\]
\[
\text{val sync : 'a event -> 'a}
\]
1. (25 points) This problem uses System F extended with addition.

(a) Give the appropriate System F typing rule for addition expressions of the form $e_1 + e_2$. (This should be easy and unrelated to the other problems.)

(b) Consider a typing context where:
   - There are no type variables in scope.
   - $x$ is the only term variable in scope and it has type $\forall \alpha. \alpha \rightarrow \alpha$.

   i. What does $\tau$ need to be for the program fragment $x \ [\tau] \ (\lambda y: \text{int}. \ y + 7) \ 11$ to typecheck? (Recall application — of types or terms — associates to the left.)
   ii. Given your choice for $\tau$, what type does $x \ [\tau] \ (\lambda y: \text{int}. \ y + 7) \ 11$ have?
   iii. Give a typing derivation for just $x \ [\tau]$ (not the entire program fragment; that’s too much work) using the typing context described above.

(c) If $v$ is an arbitrary value of type $\forall \alpha. \alpha \rightarrow \alpha$, then what might $v \ [\tau] \ (\lambda y: \text{int}. \ y + 7) \ 11$ evaluate to?

(d) If $v$ is an arbitrary value such that $v \ (\lambda y: \text{int}. \ y + 7) \ 11$ type-checks (notice $v$ is no longer polymorphic), then:
   i. What type does $v$ have?
   ii. What might $v \ (\lambda y: \text{int}. \ y + 7) \ 11$ evaluate to?

Solution:

(a)

$$
\Delta; \Gamma \vdash e_1: \text{int} \quad \Delta; \Gamma \vdash e_2: \text{int} \\
\Delta; \Gamma \vdash e_1 + e_2: \text{int}
$$

(b) i. $\tau$ must be $\text{int} \rightarrow \text{int}$
   ii. $\text{int}$
   iii.

$$
\vdash x: \forall \alpha. \alpha \rightarrow \alpha \vdash x: \forall \alpha. \alpha \rightarrow \alpha \\
\vdash \text{int} \rightarrow \text{int} \\
\vdash \text{int} \rightarrow \text{int} \\
\vdash x: \forall \alpha. \alpha \rightarrow \alpha \vdash x \ [\text{int} \rightarrow \text{int}] : (\text{int} \rightarrow \text{int}) \rightarrow (\text{int} \rightarrow \text{int})
$$

(c) It will always evaluate to 18 due to parametricity. In System F, any value of type $\forall \alpha. \alpha \rightarrow \alpha$ is totally equivalent to the identity function.

(d) i. $(\text{int} \rightarrow \text{int}) \rightarrow (\text{int} \rightarrow \tau_1)$ for any $\tau_1$. (Full credit given for $(\text{int} \rightarrow \text{int}) \rightarrow (\text{int} \rightarrow \text{int})$ because that’s what the instructor thought the answer was when he wrote the question.)
   ii. It could produce any value whatsoever. (Full credit for, “It could produce any integer.” because that’s what the instructor thought the answer was when he wrote the question.)
2. (20 points)
Consider a typed \(\lambda\)-calculus with sum types, pair types, recursive types, unit, and int.

(a) Define a type \(t_1\) for a binary tree of integers where:
- Each interior node has one integer and two children.
- Each leaf node has no data.

(b) Give a type \(t_2\) for a binary tree of integers where:
- Each node has one integer and two optional children (meaning each child may or may not be another binary tree).

(c) Explain in English how there is exactly one value of type \(t_1\) that cannot be translated to an equivalent value of type \(t_2\).

(d) Define the value you described in the previous problem using an actual \(\lambda\)-calculus expression. Make sure your value has type \(t_1\).

Solution:

(a) \(\mu \alpha. \text{unit} + (\text{int} * \alpha * \alpha)\)
(b) \(\mu \alpha. \text{int} * (\text{unit} + \alpha) * (\text{unit} + \alpha)\)
(c) The empty tree can be represented with a value of type \(t_1\) but not with \(t_2\) because every \(t_2\) has at least one int.
(d) (This answer varies depending how \(t_1\) is defined.) \(\text{roll}_{\mu \alpha. \text{unit} + (\text{int} * \alpha * \alpha)} A(())\)
3. (25 points)

Use Concurrent ML to complete an implementation of “infinite” arrays (in the sense that no index is out of bounds), without using Caml’s references or arrays. More specifically, implement the `new_array` function for this code:

(* Interface *)

```
type 'a myarray
val new_array : 'a -> 'a myarray (* Initially, every index maps to 'a *)
val set : 'a myarray -> int -> 'a -> unit (* change value in index *)
val get : 'a myarray -> int -> 'a (* return current value in index *)
```

(* Implementation *)

```
open Event
open Thread

let new_array init = (* for you *)

let rec assoc lst i default =
  match lst with
  | [] -> default
  | (j,v)::tl -> if i=j then v else assoc tl i default

let new_array init =
  let getter,setter = new_channel(), new_channel() in
  let rec server lst =
    sync (choose [wrap (receive getter)
       (fun (i,c) -> (sync (send c (assoc lst i init))); server lst);
       wrap (receive setter)
       (fun pr -> server (pr::lst))
      ])
  in
  ignore (Thread.create server []); getter,setter
```
4. (20 points)
Consider a class-based OOP language like we did in class, but suppose methods do not have implicit access to self (also known as this). More specifically:

- As usual, subclasses inherit the fields and methods of superclasses and can override methods. The method-lookup rules are the same.
- As usual, we “confuse” classes and types, so a subclass is a subtype.
- Unlike in OOP, a method in class C has to take an explicit argument of type C to access any fields/methods in its body. For example, instead of having a method like

  ```java
  int get_sum() { return self.x + self.y }
  ```

  and calling it like

  ```java
  e.get_sum()
  ```

  (assuming the method is defined in a class C with fields x and y), we instead would have to do something like

  ```java
  int get_sum(C obj) { return obj.x + obj.y }
  ```

  and call it like

  ```java
  e.get_sum(e).
  ```

So, this is less convenient than OOP (notice how e.get_sum(e) has to repeat e), but more flexible (since callers are not required to use the same e in both places).

(a) In this strange language, give an example showing how you want covariant subtyping on method arguments in order for overriding methods to use the fact that they are defined in the subclass.

(b) Show how covariant subtyping on method arguments is unsound by giving a use of your example from part (a) in which the program gets stuck even though it typechecks with covariant subtyping.

(c) In 1–3 sentences, explain why normal OOP does not have this “covariant subtyping is unsound, but contravariant subtyping is not what you want” problem.

Solution:

(a) `class C {
    int m(C obj) { return 0; }
}
class D extends C {
    int x;
    int m(D obj) { return obj.x; }
}

When overriding m in class D we want the argument (which is playing the role of self) to have type D, else we cannot access the x field. Since D ≤ C, we need covariant subtyping.

(b) `C c = new D();
c.m(new C());`

We use subsumption to assign a D object to c. The class definition of D allows m to override since we have covariant subtyping and D ≤ C. But then we call c.m(new C()), which will get stuck because the argument has no x field.

(c) In normal OOP, callers do not choose which object is bound to self. In e.m(), it must be e that is bound to self in the body of the call, so if e’s class overrides m assuming self has a subtype of what it has in the method being overridden, that assumption will always hold.
5. (6 points)
Consider a prototype-based (classless) approach to OOP as we did in class. Particularly recall how method-lookup is defined in this approach. Explain how to create an object \( obj \) such that any object derived from \( obj \) (either directly by have a parent slot holding \( obj \) or transitively) can have \( any \) method called on it without a “method not found” error occurring.

Hints:

- This is a very short problem.
- Nontermination.

Solution:
Create a loop in the parent slot, so method-lookup will never terminate for methods not found in \( obj \) or a derived object:

```
let obj = []
obj.parent = obj
```