The final exam questions can be found [here](#).

1. **[25 points] F♭P’ basics**

   (a) **[4 points] BNF grammar for F♭P’**
   
   \[c = \ldots F♭ \ldots | (c, e) | \textbf{Let} \ (x, x) = c \ \textbf{In} \ e\]
   
   \[v = \ldots F♭ \ldots | (v, v)\]

   (b) **[5 points] Additional opsem rules for F♭P’**

   \[(\text{Pair}) \quad e_1 \Rightarrow v_1 \quad e_2 \Rightarrow v_2\]
   
   \[\frac{(e_1, e_2) \Rightarrow (v_1, v_2)}{(\text{Pair Let}) \quad e \Rightarrow (v_1, v_2) \quad e' [v_1/x_1][v_2/x_2] \Rightarrow v\}
   
   \text{Let} \ (x_1, x_2) = e \ \text{In} \ e' \Rightarrow v\]

   (c) **[8 points] Macro encoding of First and Second in F♭P’**

   ```
   let fst pr = "Let (x, y) = "pr" In x"
   let snd pr = "Let (x, y) = "pr" In y"
   ```

   Macro encoding of \textit{Let} \ldots \ \textit{In} \ldots \ \textit{in} \ F♭P

   ```
   let let_in e1 e2 =
   "Let pr = "e1" In
   Let x = Fst pr In
   Let y = Snd pr In "e2
   ```

   Note that encodings may vary across submissions. However, any valid encoding should be able to work in a hypothetical F♭P/F♭P’ interpreter.
(d) [8 points] A possible F♯P′ interpreter eval function. Note that pairs can take values or expressions.

```ocaml
let rec eval e =
  match e with
  | Pair(e1, e2) ->
    Pair(eval e1, eval e2)
  | LetPair(x1, x2, e1, e2) ->
    let Pair(v1, v2) = eval e1 in
    let e2' = subst e2 v1 x1 in
    let e2'' = subst e2' v2 x2 in
    e2''
```

An alternate encoding could be to encode a pattern type, such that

```ocaml
type pattern = Var of ident | Pair of ident * ident
```

Then we can have a unified Let whose first argument has type pattern.

2. [10 points] Operational equivalence in F♯P′

(a) [6 points] Operational equivalence principle in F♯P′. Note that students may give different principles; here is one that follows from Definition 2.26 in the book

\[
\text{If } e_1 \Rightarrow (v_1, v_2), \text{ then } (\text{Let } (x_1, x_2) = e_1 \text{ In } e_2) \equiv e_2[v_1/x_1][v_2/x_2]
\]

(b) [4 points] Proof of Let \((x, y) = (\text{Fun } z \to z)(\text{True, } 2 + 1) \text{ In } (\text{If } x \text{ Then } y \text{ Else } 0) \equiv 3

\[
\text{Let } (x, y) = (\text{Fun } z \to z)(\text{True, } 2 + 1) \text{ In } (\text{If } x \text{ Then } y \text{ Else } 0)
\]

(by principle in a) \(\equiv (\text{If } x \text{ Then } y \text{ Else } 0)[\text{True}/x][3/y]

(by def. 2.26) \(\equiv 3
```

3. [20 points] Types in F♯P′

(a) [2 points] BNF grammar for TF♯P′

\[
\tau = \ldots F\ldots | (\tau, \tau)
\]

\((e, v, \text{ and } x \text{ are the same as before})

(b) [6 points] New opsem rules for TF♯P′

\[
\begin{align*}
\text{(Pair)} & \quad \Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2 \\
& \quad \Gamma \vdash (e_1, e_2) : (\tau_1, \tau_2) \\
\text{(Regular Let)} & \quad \Gamma \vdash e : \tau \quad \Gamma, x : \tau \vdash e' : \tau' \\
& \quad \Gamma \vdash \text{Let } x : \tau = e \text{ In } e' : \tau' \\
\text{(Pair Let)} & \quad \Gamma \vdash e : (\tau_1, \tau_2) \quad \Gamma, x_1 : \tau_1, x_2 : \tau_2 \vdash e' : \tau' \\
& \quad \Gamma \vdash \text{Let } (x_1, x_2) : (\tau_1, \tau_2) = e \text{ In } e' : \tau'
\end{align*}
\]
(c) [8 points] A possible typechecker (that distinguishes Let and LetPair)

\[
\begin{align*}
\text{let rec } & \text{tc gamma } e = \\
& \quad \text{match } e \text{ with} \\
& \quad | \text{Pair}(e_1, e_2) = \\
& \quad \quad \text{let } t_1 = \text{tc gamma } e_1 \text{ in} \\
& \quad \quad \text{let } t_2 = \text{tc gamma } e_2 \text{ in} \\
& \quad \quad \text{TPair}(t_1, t_2) \\
& \quad | \text{LetPair}(x_1, x_2, t, e_1, e_2) \rightarrow \\
& \quad \quad \text{let } t' = \text{tc gamma } e_1 \text{ in} \\
& \quad \quad \text{if equals } t \text{ } t' \\
& \quad \quad \quad \text{then} \\
& \quad \quad \quad \quad \text{let } \text{TPair}(t_1, t_2) = t' \text{ in} \\
& \quad \quad \quad \quad \text{tc } @\emptyset ((x_1, t_1) :: ((x_2, t_2) :: \text{gamma})) \text{ e_2} \\
& \quad \quad \text{else} \\
& \quad \quad \quad \quad \text{raise TypeError} \\
& \quad | \text{Let}(x, t, e_1, e_2) \rightarrow \\
& \quad \quad \text{let } t' = \text{tc gamma } e_1 \text{ in} \\
& \quad \quad \text{if equals } t \text{ } t' \\
& \quad \quad \quad \text{then} \\
& \quad \quad \quad \quad \text{tc } @\emptyset ((x, t') :: \text{gamma}) \text{ e_2} \\
& \quad \quad \text{else} \\
& \quad \quad \quad \quad \text{raise TypeError}
\end{align*}
\]

(d) [4 points] New subtyping rules: Since pairs can be treated as fixed-length records, we will need to apply similar subtyping rules to pairs. No new subtyping rules are needed if we encode pairs as records, but if we are implementing pairs directly (as in F♭P') we will need a new pair rule.

\[
(\text{Sub-Pair}) \quad \tau_1 <: \tau'_1, \tau_2 <: \tau'_2 \Rightarrow (\tau_1, \tau_2) <: (\tau'_1, \tau'_2)
\]

4. [15 points] Operational semantics of printing

(a) [8 points] Opsem rule for F♭Print. The important thing to realize is, unlike with state and actors, print statements must be ordered, so they need to be accumulated in a list. Here, L is our (ordered) list, v :: L appends v to the end of L, and L_1 @@ L_2 appends L_2 after L_1. (Note that students may use angle bracket notation).

\[
\begin{align*}
(\text{Print}) & \quad e \xrightarrow{L} v \\
\frac{\text{Print}(e) \xrightarrow{v::L} -}{-} \\
(\text{Plus}) & \quad e_1 \xrightarrow{L_1} v_1 \quad e_2 \xrightarrow{L_2} v_2 \quad v_1 + v_2 = v_3 \\
\frac{e_1 + e_2 \xrightarrow{L_1@@L_2} v_3}{-}
\end{align*}
\]

3
(b) [7 points] In AF∅V, we can add printing by constructing a pair \((L, S)\), where \(L\) is our list of print statements and \(S\) is our global actor set. That way we can preserve the ordering of print statements while preserving the lack of order in the set of actors and messages.
5. [15 points] Joey and the Y-combinator

Note that student answers may vary, so it is best to check them using the F# interpreter. (Ideally partial credit should be given for code that is conceptually correct, e.g. does function argument reversal, but is syntactically wrong.)

(a) [5 points] joeY internally reverses the arguments of the given function.

\[
\text{joeY = (Let f -> Let wrapper = Fun this -> Fun arg -> (Fun this' -> Fun arg' -> f arg' this') (this this) arg In wrapper wrapper)}
\]

(b) [5 points] joeyFix is a function that reverses arguments.

\[
\text{joeyFix = (Fun f -> Fun this' -> Fun arg' -> f arg' this')}\]

(c) [5 points] joeYY is the same as almostY on page 35 of the book.

\[
\text{joeYY = Fun body -> body body}\
\]

6. [12 points] Types in TF\#mR

(a) [6 points] Type rules for TF\#mR (Note: no penalty for writing \(\tau\) Ref over \(\tau\))

\[
\begin{align*}
\text{(Record) } & \Gamma \vdash e_1 : \tau_1 \ldots \Gamma \vdash e_n : \tau_n \\
\text{(Projection) } & \Gamma \vdash \{l_1 : e_1, \ldots; l_n : e_n\} : \{l_1 : \tau_1, \ldots; l_n : \tau_n\} \\
\text{(Mutate) } & \Gamma \vdash e : \{l_1 : \tau_1, \ldots; l_n : \tau_n\} \text{ for } i \in \{1, \ldots, n\} \\
\end{align*}
\]

(b) [3 points] Subtype or supertype: The answer is subtype. Since the non-record types cannot change under our rules (even if we mutate values), the usual subtyping rules follow, so

\[
\{a : \text{Int}, b : \text{Int}\} \prec \{a : \text{Int}\}
\]

(c) [3 points] Subtype or supertype: The answer is neither. In STF\#R, it is obvious that

\[
\{c : \{a : \text{Int}; b : \text{Int}\}\} \prec \{c : \{a : \text{Int}\}\}
\]

However, if in STF\#mR we mutate the RHS inner record to add additional fields, we get something like

\[
\{c : \{a : \text{Int}; b : \text{Int}\}\} \succ \{c : \{a : \text{Int}, b : \text{Int}, c : \text{Int}\}\}
\]

as the LHS is now a supertype of the RHS.
7. [15 points] Operational equivalence in AF\(\Omega\)V

(a) [6 points] Two expressions in AF\(\Omega\)V are equivalent iff they both evaluate to a value given the same initial global state.

(b) [9 points] It is possible to find \(e_1\) and \(e_2\) that are equivalent in F\(\Omega\) but not in AF\(\Omega\)V.

Let \(e_1 = f ; g ; 1\) and \(e_2 = f ; g ; g ; 1\), where in the context \(\mathcal{C}\) of AF\(\Omega\)V, \(f\) sends a message to an actor and \(g\) receives a message from the same actor. It is possible for \(e_1 \equiv e_2\) in F\(\Omega\) but not in AF\(\Omega\)V since the second \(g\) in \(e_2\) will not receive a message in AF\(\Omega\)V, so \(e_2\) diverges.