PL I Final Exam Solutions

Kelvin Qian

May 11, 2020

The final exam questions can be found here.

1. [25 points] F♭P' basics

   (a) [4 points] BNF grammar for F♭P'

   $$e = \ldots F\flat \ldots \,|\, (e, e) \,|\, \texttt{Let}\ (x, x) = e\ \texttt{In}\ e$$
   $$v = \ldots F\flat \ldots \,|\, (v, v)$$

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

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

   (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 `Let ... In ...` 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.

```
let rec eval e =
    match e with
    (* standard Fb expressions, including normal Let *)
    | 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

```
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

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

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

$$\begin{array}{rl} & \texttt{Let }\ (x, y) = (\texttt{Fun }\ z \to z)(\texttt{True}, 2+1)\ \texttt{ In }\ (\texttt{If }\ x\ \texttt{ Then }\ y\ \texttt{ Else }\ 0) \\ \text{(by principle in a)} & \cong (\texttt{If }\ x\ \texttt{ Then }\ y\ \texttt{ Else }\ 0)[\texttt{True}/x][3/y] \\ \text{(by def. 2.26)} & \cong 3 \end{array}$$

3. [20 points] Types in F♭P'

(a) [2 points] BNF grammar for TF♭P'

$$\tau = \ldots F\flat \ldots \mid (\tau, \tau)$$

(e, v, and x are the same as before)

(b) [6 points] New opsem rules for TF♭P'

$$\text{(Pair)}\quad \frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash (e_1, e_2) : (\tau_1, \tau_2)}$$

$$\text{(Regular Let)}\quad \frac{\Gamma \vdash e : \tau \quad \Gamma, x : \tau \vdash e' : \tau'}{\Gamma \vdash \texttt{Let }\ x : \tau = e\ \texttt{ In }\ e' : \tau'}$$

$$\text{(Pair Let)}\quad \frac{\Gamma \vdash e : (\tau_1, \tau_2) \quad \Gamma, x_1 : \tau_1, x_2 : \tau_2 \vdash e' : \tau'}{\Gamma \vdash \texttt{Let }\ (x_1, x_2) : (\tau_1, \tau_2) = e\ \texttt{ In }\ e' : \tau'}$$

(c) [8 points] A possible typechecker (that distinguishes `Let` and `LetPair`)

```
let rec tc gamma e =
    match e with
    (* standard Fb expressions *)
    | Pair(e1, e2) =
        let t1 = tc gamma e1 in
        let t2 = tc gamma e2 in
        TPair(t1, t2)
    | LetPair(x1, x2, t, e1, e2) ->
        let t' = tc gamma e1 in
        if equals t t'
        then
            let TPair(t1, t2) = t' in
            tc @@ ((x1, t1) :: ((x2, t2) :: gamma)) e2
        else
            raise TypeError
    | Let(x, t, e1, e2) ->
        let t' = tc gamma e1 in
        if equals t t'
        then
            tc @@ ((x, t') :: gamma) e2
        else
            raise TypeError
```

(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.

$$\text{(Sub-Pair)} \quad \frac{\tau_1 <: \tau_1' \quad \tau_2 <: \tau_2'}{(\tau_1, \tau_2) <: (\tau_1', \tau_2')}$$

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).

$$\text{(Print)} \quad \frac{e \overset{L}{\Longrightarrow} v}{\texttt{Print}(e) \overset{v::L}{\Longrightarrow} \_}$$

$$\text{(Plus)} \quad \frac{e_1 \overset{L_1}{\Longrightarrow} v_1 \quad e_2 \overset{L_2}{\Longrightarrow} v_2 \quad v_1 + v_2 = v_3}{e_1 + e_2 \overset{L_1 @@ L_2}{\Longrightarrow} v_3}$$

(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, eg. does function argument reversal, but is syntactically wrong.)

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

```
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.

```
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.

```
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 τ `Ref` over τ)

$$\text{(Record)}\quad \frac{\Gamma \vdash e_1 : \tau_1 \quad \ldots \quad \Gamma \vdash e_n : \tau_n}{\Gamma \vdash \{l_1 = e_1; \ldots; l_n = e_n\} : \{l_1 : \tau_1; \ldots; l_n : \tau_n\}}$$

$$\text{(Projection)}\quad \frac{\Gamma \vdash e : \{l_1 : \tau_1; \ldots; l_n : \tau_n\}}{\Gamma \vdash e.l_i : \tau_i \text{ for } i \in \{1, \ldots, n\}}$$

$$\text{(Mutate)}\quad \frac{\Gamma \vdash e : \{l_1 : \tau_1; \ldots; l_n : \tau_n\} \quad \Gamma \vdash e' : \tau_i}{\Gamma \vdash e.l_i \texttt{ <- } e' : \texttt{TUnit} \text{ for } i \in \{l, \ldots, n\}}$$

(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

$$\{\texttt{a: Int, b: Int}\} <: \{\texttt{a: Int}\}$$

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

$$\{\texttt{c: }\{\texttt{a: Int; b: Int}\}\} <: \{\texttt{c: }\{\texttt{a: Int}\}\}$$

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

$$\{\texttt{c: }\{\texttt{a: Int; b: Int}\}\} :> \{\texttt{c: }\{\texttt{a: Int, b: Int, c: Int}\}\}$$

as the LHS is now a supertype of the RHS.

7. [15 points] Operational equivalence in AF♭V

(a) [6 points] Two expressions in AF♭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♭ but not in AF♭V.
Let $e_1 = f; g; 1$ and $e_2 = f; g; g; 1$, where in the context $\texttt{C}$ of AF♭V, f sends a message to an actor and g receives a message from the same actor. It is possible for $e_1 \cong e_2$ in F♭ but not in AF♭V since the second g in e_2 will not receive a message in AF♭V, so e_2 diverges.