Jason Zhang jzhan127

PL Final

1.

a. $e ::= v \mid \text{Fb stuff} \ldots \mid \overset{(\text{pair})}{(e, e)} \mid \text{Let } (x, y) = e \text{ In } e$

$v ::= x \mid y \mid \ldots \text{ Fb Stuff} \ldots \mid (v, v)$

$x ::=$ (a | b | ... all other Fb variable names)

$y ::=$ (a | b | ... all other Fb variable names)

b. Pair rule:

$$\frac{e_1 \Rightarrow v_1 \quad , \quad e_2 \Rightarrow v_2}{(e_1, e_2) \Rightarrow (v_1, v_2)}$$

ExtensionLet rule:

$$\frac{e_1 \Rightarrow (v_1, v_2) \quad , \quad (e_2[v_1/x])[v_2/y]}{\text{Let } (x, y) = e_1 \text{ In } e_2 \Rightarrow v_3}$$

C. FbP' and FbP have similar expressiveness:

FbP' left: (Fun m → Fun p → Let (x,y) = p In (If m Then x Else y)) True (i, j)

FbP' right: (Fun m → Fun p → Let (x, y) = p In (If m Then x Else y)) False (i, j)

FbP match: (Fun e → Fun p → e (left p) (right p)) e_1 (i, j) → for input pair (i, j)

e_1 takes in both values in a pair i.e.: Fun x → Fun y → e (some expression that matches on either x or y)

1. d.

*LetM is our Let extension for matching

```
type expr = ... Fb stuff | Pair of expr * expr
|LetM of ident * ident * expr * expr
```

```
let rec eval e =
    if check_closed e = true Then
        match e with
        |... Fb stuff
        | Pair(x,y) -> Pair(eval x, eval y)
        |LetM(x,y,e1,e2) -> (match eval e1 with
                             | Pair(v1,v2) -> eval subst x y v1 v2 e2)
                             | _ -> raise Error
        | _ -> raise Error
    Else
        raise NotClosed
```

* check_closed returns bool if expression is closed or not

* subst substitutes x y for $v_1 v_2$ in e_2

2 a.

$\text{Let } (x,y) = e_1 \text{ In } e_2 \cong (e_2[v_1/x])[v_2/y]$

provided

e_1 evaluates to a pair: (v_1,v_2)

b. Using the above principle we want to prove:

Let (x,y) = (Fun z → z)(True, 2+1) In If x Then y Else 0 $\cong$ 3

looking at (Fun z→z)(True, 2+1) this evaluates to (True, 2+1) which satisfies our pair requirement for e_1 in the above rule. Thus we get:

Let (x,y) = (Fun z→z)(True, 2+1) In If x then y Else 0 $\cong$ If True Then (2+1) Else 0

which evaluates to 2+1. from our sum rule, 2+1 $\cong$ 3. Thus by transitivity: Let (x,y) = (Fun z→z)(True, 2+1) In If x Then y Else 0 $\cong$ 3

*LetM is our Let e

3.

a. TFbP': $e ::= v \mid e + e \mid e - e \mid e \text{ And } e \mid e \text{ Or } e \mid \text{Not } e \mid e = e \mid e\, e \mid$
$\text{If } e \text{ then } e \text{ Else } e \mid \text{Let } x : \tau = e \text{ in } e \mid \text{Let Rec } f\, x : \tau = e : \tau \text{ In } e \mid$
$\text{Let } (x, y) : (\tau, \tau) = e \text{ in } e$

$v ::=$ regular Fb stuff... $\mid \text{Fun } x : \tau \to e \mid (v, v) \mid x \mid y \mid$
$x ::=$ Fb variable names... $\mid$
$y ::=$ Fb variable names... $\mid$
$\tau ::= \text{Bool} \mid \text{Int} \mid \tau \to \tau \mid (\tau, \tau)$

b. ... omitted ones alredy given in the book

Let (reguler):
$$\frac{\overline{\Gamma, \vdash e_1 : \tau_1} \qquad \overline{\Gamma, x : \tau \vdash e_2 : \tau''}}{\Gamma \vdash (\text{Let } x : \tau = e_1 \text{ In } e_2) : \tau''}$$

Let (pairs):
$$\frac{\overline{\Gamma, \vdash e_1 : (\tau, \tau'')} \qquad \overline{\Gamma, (x, y) : (\tau, \tau'') \vdash e_2 : \tau'''}}{\Gamma \vdash (\text{Let } (x, y) : (\tau, \tau'') = e_1 \text{ In } e_2) : \tau'''}$$

Pairs:
$$\frac{\overline{\Gamma \vdash e_1 : \tau} \qquad \overline{\Gamma \vdash e_2 : \tau''}}{\Gamma \vdash (e_1, e_2) : (\tau, \tau'')}$$

c.
```
let rec typecheck gamma e =
   match e with
   |... old TFb stuff
   | Let(id, type, e1, e2) -> (if (typecheck gamma e1) = type Then
                                    typecheck ((id, type)::gamma) e2
                              Else
                                    raise Type Error)
   | Let(x, y, type1, type2, e1, e2) -> (match typecheck gamma e1 with
       | (type1, type2) -> typecheck ((x, type1):: (y, type2)::gamma) e2
       | _ -> raise type Error)
   | Pairs(e1, e2) -> Pairs(typecheck gamma e1, typecheck gamma e2)
   | _ -> raise Error
```

| Else

3 d. (Sub pairs):

$$\frac{\tau_1 <: \tau_1' \qquad \tau_2 <: \tau_2'}{(\tau_1, \tau_2) <: (\tau_1', \tau_2')}$$

4 a. * l denotes list of side effects from an operation (list keeps ordering)

FbP sum rule: (*other rules follow same way)

$$\frac{e_1 \overset{l_1}{\Longrightarrow} v_1 \ , \ e_2 \overset{l_2}{\Longrightarrow} v_2 \ , \ v_1, v_2 \in \mathbb{Z}}{e_1 + e_2 \overset{l_1 @ l_2}{\Longrightarrow} v_1 + v_2}$$ (integer sum)

FbP print:

$$\frac{e \overset{l}{\Rightarrow} v \qquad v \in \mathbb{Z}}{\text{Print}(e) \overset{l@[\text{Print}(v)]}{\Longrightarrow} v}$$

b. define $\langle S, l \rangle$ where $\underline{S}$ is the original global soup of AFbV and $\underline{l}$ is the list of print side effects (to keep print ordering)

(+ Rule):

$$\frac{e_1 \overset{\langle S, l\rangle}{\Rightarrow} v_1 \ , \ e_2 \overset{\langle S', l'\rangle}{\Rightarrow} v_2 \text{ where } v_1, v_2 \in \mathbb{Z}}{e_1 + e_2 \overset{\langle S \cup S', l@l' \rangle}{\Longrightarrow} v_1 + v_2}$$

(Send Rule):

$$\frac{e_1 \overset{\langle S, l\rangle}{\Rightarrow} a \ , \ e_2 \overset{\langle S', l'\rangle}{\Rightarrow} v}{e_1 \leftarrow e_2 \overset{\langle S \cup S' \cup \{[a \leftarrow v]\}, l@l' \rangle}{\Longrightarrow} v}$$

examples of how to also include Print(e) side effects

5.

a. joeY : (Fun code -> Let repl = Fun this -> Fun arg ->
code arg (this this) In repl repl)

b. joeyFix : (Fun code => Fun this -> Fun arg -> code arg this)

c. joeYY : (Fun code -> code code)

6.

a. ... omitting TFb type rules

Record update:

$$\frac{\Gamma \vdash e : \{l_1 : \tau_1 ; \dots ; l_n : \tau_n\},\ \Gamma \vdash e' : \tau_i}{\Gamma \vdash e.l_i \leftarrow e' : \tau_i \text{ for } 1 \le i \le n}$$

Record rule:
$$\frac{\Gamma \vdash e_1 : \tau_1 \dots \Gamma \vdash e_n : \tau_n}{\Gamma \vdash \{l_1 = e_1 ; \dots ; l_n = e_n\} : \{l_1 = \tau_1 ; \dots ; l_n = \tau_n\}}$$

Projection rule:
$$\frac{\Gamma \vdash e : \{l_1 : \tau_1 ; \dots ; l_n : \tau_n\}}{\Gamma \vdash e.l_i : \tau_i \text{ for } 1 \le i \le n}$$

b. Correct direction:

$\underline{\{a : Int ; b : Int\} <: \{a : Int\}}$ Due to the fact that in the new STFbmR, mutable records can only change the value of an existing record entry as long as the new value is of the same type. This means when we update values in a record, the types in the Record remain the same. Thus the old subtyping rules in STFbR will still hold in which the record with fewer elements is the supertype. (Provided the subtype record has at least all the entries in the supertype record).

6 c. $\{c: \{a: Int; b: Int\}\} <: \{c: \{a: Int\}\}$

For the same reason in 6b. we can safely apply similar subtype reasoning as in STFbR. $\{c: \{a: Int\}\}$ has fewer entries than $\{c: \{a: Int; b: Int\}\}$ and $\{c: \{a: Int; b: Int\}\}$ has all the entries in $\{c: \{a: Int\}\}$, thus the above relation must be the correct one.

7.

a. $C ::= \bullet \mid \ldots$ AFbV stuff...

we ~~define~~ define operational equivalence for AFbV as:

$e \cong e'$ if and only if for all contexts C such that $C[e]$ and $C[e']$ are closed, $C[e] \overset{S}{\Longrightarrow} v$ if and only if $C[e'] \overset{S'}{\Longrightarrow} v'$, where The global Soup S is reasonably equivalent to S'.

we define "reasonably equivalent" similarly to the Leibnitzian notion in that we see no discernable difference in execution between the two arguments. In this case, we define reasonable equivalence between S and s' in AFbV to be: S is reasonably equivalent to S' if $\forall \langle a, v\rangle$ with a corresponding $[a \leftarrow msg]$ in S, $\exists$ $\langle a, v\rangle$ and corresponding $[a \leftarrow msg]$ in S' and $\forall$ send operations in S, $\exists$ the same send operation in S' (vice versa) and the # of send operations in S = # send operations in S'. We basically expect for every actor in S with at least 1 send operation to it, there must be the same actor and send operation in S' and the message operations and # of message operations in S must be the same in S'. This definition encompasses the example in the problem where we had an unused actor since unused actors have no messages sent to it, therefore S' does not need to include actors in S that have no corresponding messages to them to be reasonably equivalent to S. (We violated none of the above properties)

7b.

There does not exist a pair e1/e2 that satisfies the claim. Given that in AFbV, we do not read from the global state we only write to it, and that concrete Fb programs do not have any extra side effects, any expression from Fb applied to an AFbV context, will generate reasonable equivalent global stores S and S' and thus they will be operationally equivalent in AFbV. AFbV programs generate side effects that do not require ordering thus expressions with commutative properties like + in Fb that are operationally equivalent in Fb will hold in AFbV since AFbV is agnostic to ordering as well.