OPEN DISTRIBUTED SYSTEMS

- Addition of new components.
- Replacement of existing components.
- Changes in interconnections.

ACTOR CONFIGURATIONS

model open system components:

- set of individually named actors.
- messages "en-route".
- interface to environment:
  * receptionists
  * external actors

SYNCHRONOUS VS ASYNCHRONOUS COMMUNICATION

- π-Calculus (and other process algebras such as CCS, CSP) take synchronous communication as a primitive.

- Actors assume asynchronous communication is more primitive.

COMMUNICATION MEDIUM

- In π-Calculus, channels are explicitly modelled. Multiple processes can share a channel, potentially causing interference.

- In the actor model, the communication medium is not explicit. Actors (active objects) are first-class, history-sensitive entities with an explicit identity used for communication.

FAIRNESS

The actor model theory assumes fair computations:

1. message delivery is guaranteed.
2. individual actor computations are guaranteed to progress.

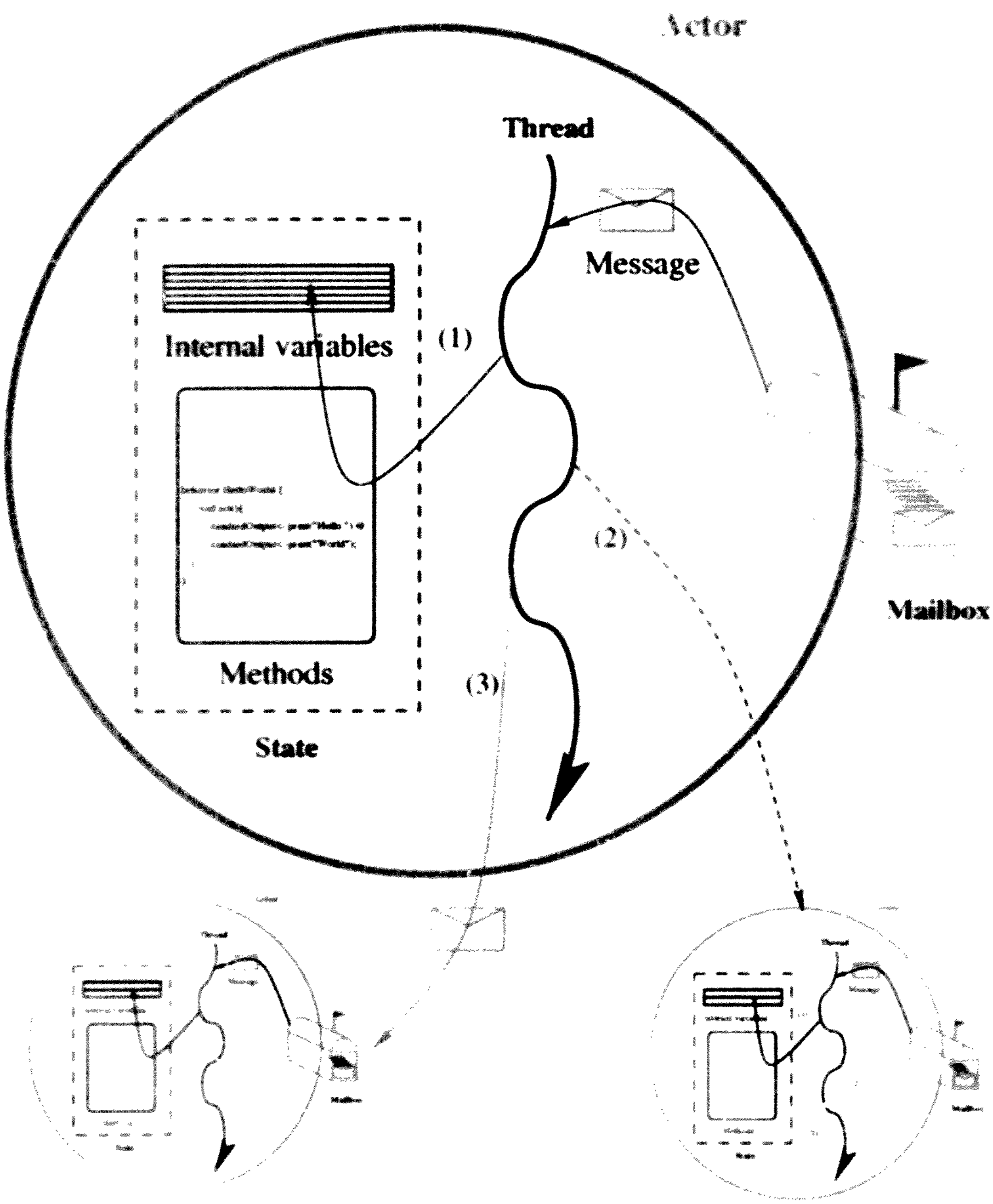
Fairness is very useful for reasoning about equivalences of actor programs but can be hard/expensive to guarantee; in particular when distribution and failures are considered.

PROGRAMMING LANGUAGES INFLUENCED BY π-CALCULUS AND ACTORS.

- Scheme '75
- Act1 '87
- Acore '87
- Rosette '89
- Obliq '94
- Erlang '93
- ABCL '90
- SALSA '99

- Amber '86
- Facile '89
- CML '91
- Pict '94
- Nomadic Pict '99
- JOCAML '99

Actor (Agent) Model



AGHA, MASON, SMITH & TALCOTT

1. - Extend a functional language (λ-calculus + ifs + pairs) with actor primitives

2. - Define an operational semantics for actor configurations.

3. - Study various notions of equivalence of actor expressions and configurations.

4. - Assume fairness:
   - guaranteed message delivery.
   - individual actor progress.

λ-CALCULUS

SYNTAX

$$
\begin{aligned}
e ::= &\ v \\
\mid &\ \lambda v.e \\
\mid &\ (e\ e)
\end{aligned}
$$

value
function abstracti
applicati

EXAMPLE

$$((\lambda x.\underbrace{x}_{}) \ 5)$$

5

$x\{5/x\}$ ← π

$[5/x]x$ ⇐ ?

$pr(x,y)$ returns a pair containing x & y.

$ispr(x)$ returns t if x is a pair; f otherwise

$1st(pr(x,y)) = x$ 1st returns the first value of a pair

$2nd(pr(x,y)) = y$ 2nd returns the second value.

ACTOR PRIMITIVES

send(a,v)

sends value v to actor a.

new(b)

creates a new actor with behavior b and returns the identity/name of the newly created actor.

ready(b)

becomes ready to receive a new message with behavior b.

ACTOR LANGUAGE EXAMPLE

$$b5 = rec(\lambda y. \lambda x. seq(send(x, 5), ready(y)))$$

receives an actor name x and sends the number 5 to that actor, then it becomes ready to process new messages with the same behavior y.

SAMPLE USAGE

$$send(new(b5), a)$$

A SINK

$$sink = rec(\lambda b. \lambda m. ready(b))$$

an actor that disregards all messages.

REFERENCE CELL IN ACTOR LAMBDA


```
cell = rec (λb. λc. λm.
   if (get?(m),
         seq( send(cust(m), c),
              ready(b(c))),
         if (set?(m),
              ready(b(contents(m))),
              ready(b(c)))))
```

Using the cell:

```
let a = new(cell(0)) in
   seq( send(a, mkset(7)),
        send(a, mkset(2)),
        send(a, mkget(c)))
```

JOIN CONTINUATIONS

Consider:

```
treeprod = rec(λf. λtree.
                  if(isnat(tree),
                     tree,
                     f(left(tree)) * f(right(tree))
                    ))
```

which multiplies all leaves of a tree, which are numbers.

You can do the "left" and "right" computations concurrently.

TREE PRODUCT BEHAVIOR

$B_{treeprod}$ =

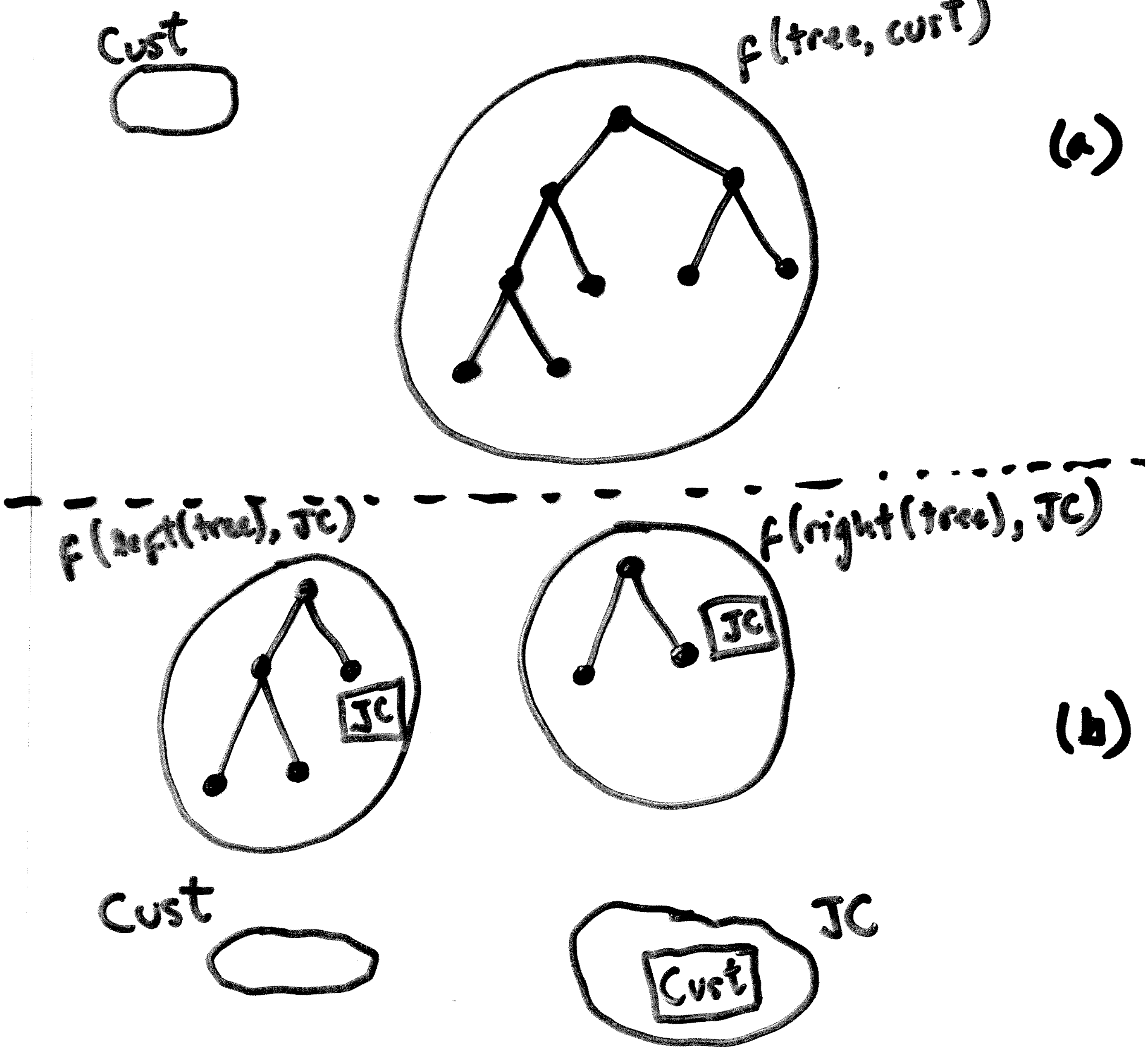
```
rec(λb. λself. λm.
    seq( become(b(self)),
         if(isnat(tree(m)),
            send(cust(m), tree(m)),
            letactor{newcust := B_joincont
                                 (cust(m), ∅, nil)}
              seq(send(self,
                    pr(left(tree(m)), newcust)),
                  send(self,
                    pr(right(tree(m)), newcust)))
         )
    )
)
```

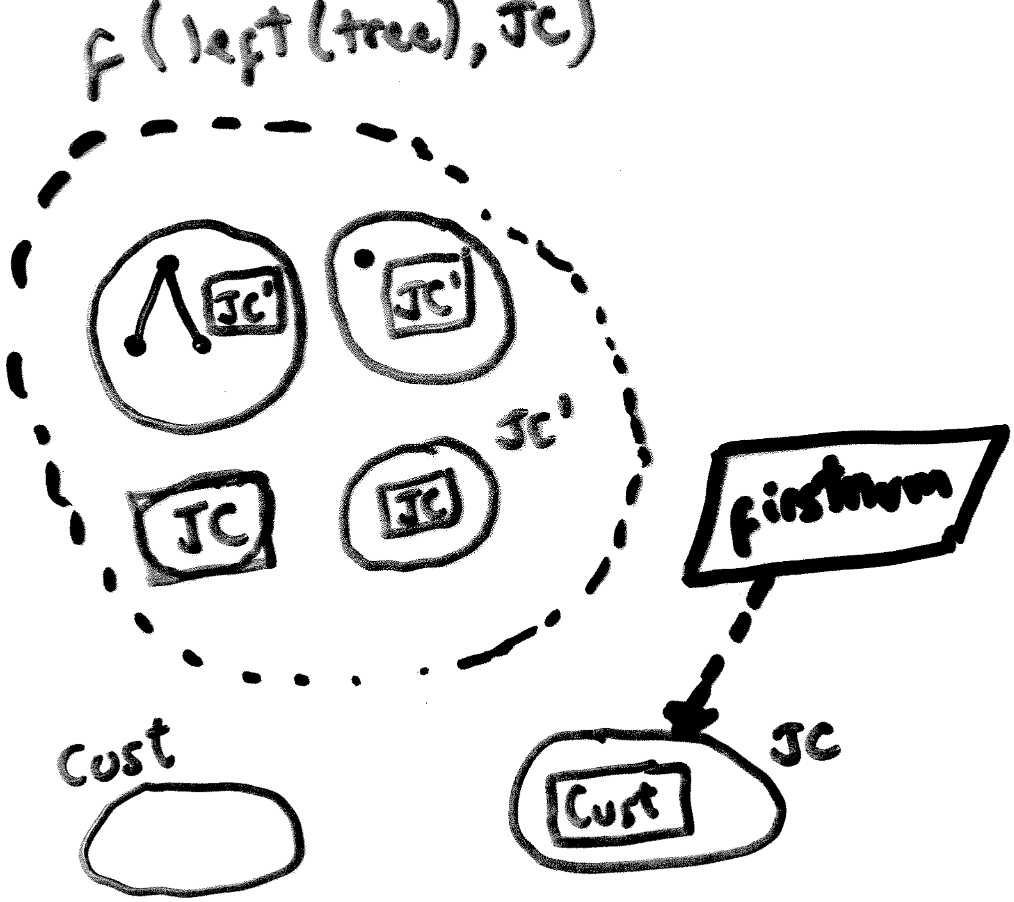
TREE (continued)

$B_{joincont}$ =

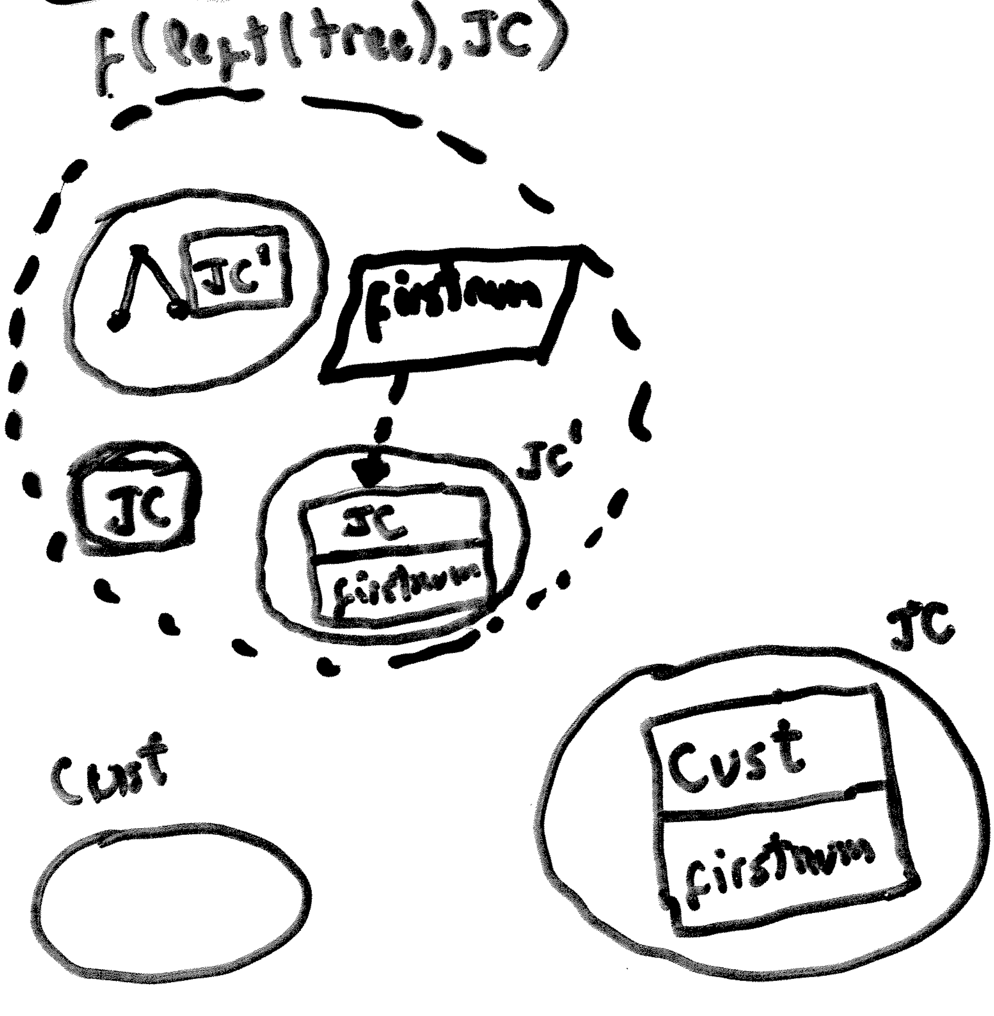
```
rec (λb. λcust, λnargs, λfirstnum, λnum
        if (eq (nargs, 0),
            become (b (cust, ⊥, num)),
            seq ( become (sink),
                  send (cust, firstnum * num))))
```

SAMPLE EXECUTION

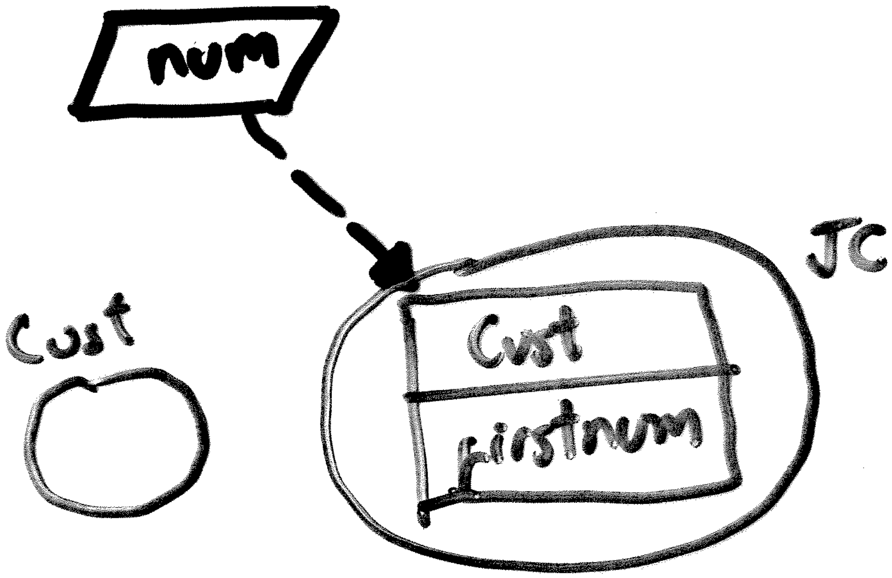



f(left(tree), JC)
JC'
JC'
JC'
JC
JC
firstnum
Cust
JC
Cust

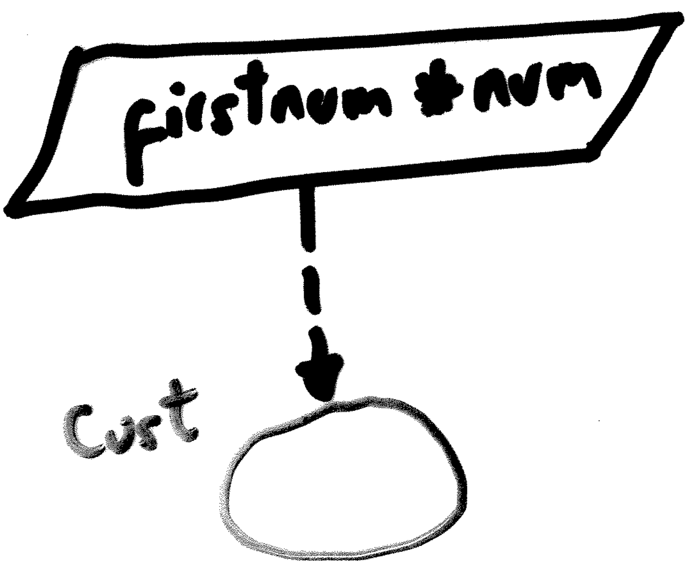
(c)


f(left(tree), JC)
JC'
firstnum
JC'
JC
JC
firstnum
JC
Cust
Cust
firstnum

(d)



(e)



(f)

DINING PHILOSOPHERS IN ACTOR LANGUAGE

```
phil = rec (λb. λl. λr. λself. λsticks. λm.
   if (eq? (sticks, 0),
        ready (b(l, r, self, 1)),
        seq ( send (l, mkrelease (self)),
              send (r, mkrelease (self)),
              send (l, mkpickup (self)),
              send (r, mkpickup (self)),
              ready (b(l, r, self, 0)))))
```

DINING PHILOSOPHERS IN ACTOR LANGUAGE (2)

```
chopstick = rec(λb. λh. λw. λm.
    if(pickup?(m),
        if(eq?(h, nil),
            seq(send(getphil(m), nil),
                ready(b(getphil(m), nil))),
            ready(b(h, getphil(m)))),
        if(release?(m),
            if(eq?(w, nil),
                ready(b(nil, nil)),
                seq(send(w, nil),
                    ready(b(w, nil)))),
            ready(b(h, w)))))
```

DINING PHILOSOPHERS IN ACTOR LANG. (?)


```
letrec c1 = new(chopstick(nil, nil)),
       c2 = new(chopstick(nil, nil)),
       p1 = new(phil(c1, c2, p1, 0)),
       p2 = new(phil(c2, c1, p2, 0)) in e
```

where e is defined as:

```
e = seq(send(c1, mkpickup(p1)),
        send(c2, mkpickup(p1)),
        send(c1, mkpickup(p2)),
        send(c2, mkpickup(p2)))
```

DINING PHILOSOPHERS IN ACTOR LANG (4)

Auxiliary definitions:

mkpickup = $\lambda p.p$

mkrelease = nil

pickup? = $\lambda m.$ not (eq?(m, nil))

release? = $\lambda m.$ eq?(m, nil)

getphil = $\lambda m.m$

OPERATIONAL SEMANTICS FOR ACTOR CONFIGURATIONS

$$\kappa = \langle\langle \alpha \mid \mu \rangle\rangle^{\rho}_{\chi}$$

α is a function mapping actor names (represented as variables) to actor states.

μ is a multi-set of messages "en-route".

ρ is a set of receptionists

χ is a set of external actors.

Given $A = Dom(\alpha)$:

$$\rho \subseteq A \quad , \quad A \cap \chi = \emptyset$$

- if $\alpha(a) = (?_{a'})$, then $a' \in A$
- if $a \in A$, then $FV(\alpha(a)) \subseteq A \cup \chi$.
- if $\langle v_0 \Leftarrow v_1 \rangle \in \mu$, $FV(v_0, v_1) \subseteq A \cup \chi$.

$\underline{\alpha} \in \mathbb{X} \xrightarrow{f} A_s$

$A_s = (?_x) \cup (\mathbb{V}) \cup [\mathbb{E}]$

$\underline{\mu} \in M_\omega[\mathbb{M}]$

$\mathbb{M} = \langle \mathbb{V} \Leftarrow \mathbb{V} \rangle$

$\langle a \Leftarrow s \rangle$

$\underline{\rho}, \underline{\chi} \in P_\omega[\mathbb{X}]$

$\mathbb{V} = \mathbb{A}t \cup \mathbb{X} \cup \lambda \mathbb{X}.\mathbb{E} \cup pr(\mathbb{V}, \mathbb{V})$

$\mathbb{E} = \mathbb{V} \cup app(\mathbb{E}, \mathbb{E}) \cup \mathbb{F}_n(\mathbb{E}^n)$

LABELLED TRANSITION RELATION ($\mapsto$)

$\langle \text{fun}: a \rangle$

$$e \xmapsto{\lambda}_{Dom(\alpha) \cup \{a\}} e' \quad \Rightarrow$$

$$\langle\!\langle \alpha, [e]_a \mid \mu \rangle\!\rangle^{\rho}_{\chi} \mapsto \langle\!\langle \alpha, [e']_a \mid \mu \rangle\!\rangle^{\rho}_{\chi}$$

$\langle \text{newactor}: a, a' \rangle$

$$\langle\!\langle \alpha, [R[\text{newactor}(e)]]_a \mid \mu \rangle\!\rangle^{\rho}_{\chi} \mapsto$$

$$\langle\!\langle \alpha, [R[a']]_a, [e]_{a'} \mid \mu \rangle\!\rangle^{\rho}_{\chi} \qquad a' \text{ fresh}$$

$\langle \text{send}: a, v_0, v_1 \rangle$

$$\langle\!\langle \alpha, [R[\text{send}(v_0, v_1)]]_a \mid \mu \rangle\!\rangle^{\rho}_{\chi} \mapsto$$

$$\langle\!\langle \alpha, [R[\text{nil}]]_a \mid \mu, \langle v_0 \Leftarrow v_1 \rangle \rangle\!\rangle^{\rho}_{\chi}$$

LABELLED TRANSITION RELATION ($\mapsto$) CONTINUED

$\langle \text{receive} : v_0, v_1 \rangle$

$$\langle\!\langle \alpha, [\mathrm{ready}(v)]_{v_0} \mid \langle v_0 \Leftarrow v_1 \rangle, \mu \rangle\!\rangle^{\rho}_{\chi} \mapsto$$

$$\langle\!\langle \alpha, [\mathrm{app}(v, v_1)]_{v_0} \mid \mu \rangle\!\rangle^{\rho}_{\chi}$$

$\langle \text{out} : v_0, v_1 \rangle$

$$\langle\!\langle \alpha \mid \mu, \langle v_0 \Leftarrow v_1 \rangle \rangle\!\rangle^{\rho}_{\chi} \mapsto \langle\!\langle \alpha \mid \mu \rangle\!\rangle^{\rho'}_{\chi}$$

if $v_0 \in \chi$ and $\rho' = \rho \cup (FV(v_1) \cap Dom(\alpha))$

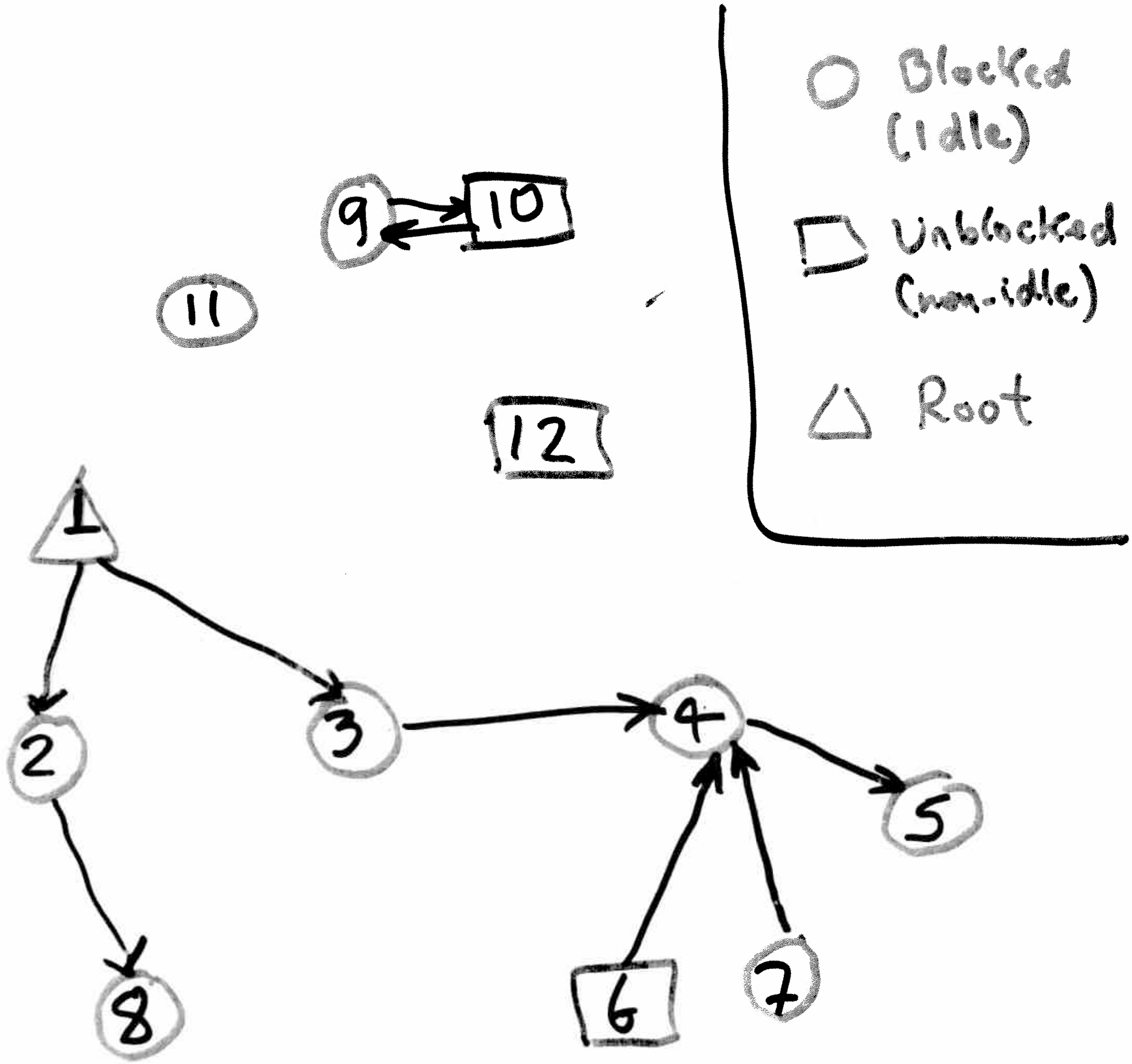
$\langle \text{in} : v_0, v_1 \rangle$

$$\langle\!\langle \alpha \mid \mu \rangle\!\rangle^{\rho}_{\chi} \mapsto \langle\!\langle \alpha \mid \mu, \langle v_0 \Leftarrow v_1 \rangle \rangle\!\rangle^{\rho}_{\chi'}$$

if $v_0 \in \rho$ and $FV(v_1) \cap Dom(\alpha) \subseteq \rho$

and $\chi' = \chi \cup (FV(v_1) - Dom(\alpha))$

ACTOR GARBAGE COLLECTION



EXERCISES

① Write get?
cust
set?
contents
mkset
mkget

to complete the reference cell example in the AMST actor language.

② Modify Bcell to notify a customer when the cell value is updated (such as in the π-calculus cell example).