COMPUTATION SEQUENCES AND PATHS

If K is a configuration, then the computation tree $\mathcal{T}(K)$ is the set of all finite sequences of labelled transitions $[K_i \xrightarrow{l_i} K_{i+1} \mid i < n]$ for some $n \in \mathbb{N}$, with $K = K_0$. Such sequences are called computation sequences.

A computation path from K is a maximal linearly ordered set of computation sequences in the computation tree, $\tau(K)$.

$\tau^\infty(K)$ denotes the set of all paths from K.

FAIRNESS

A path $\pi = [K_i \xrightarrow{l_i} K_{i+1} \mid i < \alpha]$ in the computation tree $\mathcal{T}^{\infty}(\kappa)$ is fair if each enabled transition eventually happens or becomes permanently disabled.

For a configuration K we define $\mathcal{F}(\kappa)$ to be the subset of $\mathcal{T}^{\infty}(\kappa)$ that are fair.

EQUIVALENCE OF EXPRESSIONS

Operational equivalence
Testing equivalence } Observational Equivalence

Two program expressions are said to be equivalent if they behave the same when placed in any observing context.

An observing context is a complete program with a hole, such that all free variables in expressions being evaluated become captured, when placed in the hole.

EVENTS AND OBSERVING CONTEXTS

A new event primitive operator is introduced.

The $\mapsto$ reduction relation is extended:

$\langle e: a \rangle$

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

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

An observing configuration is one of the form:

$$\langle\!\langle \alpha, [C]_a \mid \mu \rangle\!\rangle$$

where C is a hole-containing expression, or context.

OBSERVATIONS

Let K be a configuration of the extended language, and let $\pi = [K_i \xrightarrow{l_i} K_{i+1} \mid i < \infty]$ be a fair path, i.e. $\pi \in F(K)$. Define:

$$\mathrm{obs}(\pi) = \begin{cases} s & \text{if } (\exists\, i < \infty, a)(l_i = \langle e : a \rangle) \\ f & \text{otherwise} \end{cases}$$

$$\mathrm{Obs}(K) = \begin{cases} s & \text{if } (\forall \pi \in F(K))(\mathrm{obs}(\pi) = s) \\ sf & \text{otherwise} \\ f & \text{if } (\forall \pi \in F(K))(\mathrm{obs}(\pi) = f) \end{cases}$$

EQUIVALENCE EXAMPLE

$$e_1 = \mathrm{send}(a, 1)$$

$$e_2 = \mathrm{send}(a, 2)$$

$$e_3 = \mathrm{seq}(\mathrm{send}(a,1), \mathrm{send}(a,2))$$

$$e_4 = \mathrm{seq}(\mathrm{send}(a,2), \mathrm{send}(a,1))$$

$$O = \emptyset, [\mathrm{ready}(\lambda n.\, \mathrm{if}(n=1, \mathrm{event}(), \mathrm{ready}(\mathrm{sink})))]_a, [\square]_{a'} \parallel \emptyset$$

$$O' = \emptyset, [\mathrm{ready}(\lambda n.\, \mathrm{if}(n=2, \mathrm{event}(), \mathrm{ready}(\mathrm{sink})))]_a, [\square]_{a'} \parallel \emptyset$$

$$O^* = \emptyset, [\mathrm{ready}(\lambda n.\, \mathrm{if}(n=1, \mathrm{send}(a^*, \mathrm{nil}), \mathrm{ready}(\mathrm{sink})))]_a, [\square]_{a'}, [\mathrm{ready}(\lambda b.\, \mathrm{if}(b=\mathrm{true}, \mathrm{event}(), \mathrm{ready}(\mathrm{sink})))]_{a^*} \parallel \langle a^* \Leftarrow \mathrm{true} \rangle$$

$$\mathrm{Obs}\,(0 \blacktriangleright e_1 \blacktriangleleft) =$$

$$\mathrm{Obs}\,(0 \blacktriangleright e_2 \blacktriangleleft) =$$

$$\mathrm{Obs}\,(0 \blacktriangleright e_3 \blacktriangleleft) =$$

$$\mathrm{Obs}\,(0 \blacktriangleright e_4 \blacktriangleleft) =$$

THREE EQUIVALENCES

The natural equivalence is equal observations are made in all closing configuration contexts.

Other two equivalences (weaker) arise if sf observations are considered as good as s observations; or if sf observations are considered as bad as f observations.

TESTING OR CONVEX OR PLOTKIN OR EGLI-MILNER

$$e_0 \cong_1 e_1 \text{ iff } \forall O: \left(\mathrm{Obs}(O[e_0]) = \mathrm{Obs}(O[e_1])\right)$$

MUST OR UPPER OR SMYTH

$$e_0 \cong_2 e_1 \text{ iff } \forall O: \left(\mathrm{Obs}(O[e_0]) = s \iff \mathrm{Obs}(O[e_1]) = s\right)$$

MAY OR LOWER OR HOARE

$$e_0 \cong_3 e_1 \text{ iff } \forall O: \left(\mathrm{Obs}(O[e_0]) = f \iff \mathrm{Obs}(O[e_1]) = f\right)$$

CONGRUENCE

$$e_0 \cong_j e_1 \Rightarrow C[e_0] \cong_j C[e_1] \quad \text{for } j=1,2,3$$

By construction, all equivalences defined are congruences.

PARTIAL COLLAPSE

			e_1	
		s	sf	f
	S	✓	✗	✗
e_0	SF	✗	✓	✗
	F	✗	✗	✓

$\cong_1$

			e_1	
		s	sf	f
	S	✓	✗	✗
e_0	sf	✗	✓	*
	f	✗	*	✓

$\cong_2$

			e_1	
		s	sf	F
	S	✓	✓	✗
e_0	SF	✓	✓	✗
	F	✗	✗	✓

$\cong_3$

(1=2) $e_0 \cong_1 e_1$ iff $e_0 \cong_2 e_1$ (due to fairness)

(1 ⇒ 3) $e_0 \cong_1 e_1$ implies $e_0 \cong_3 e_1$

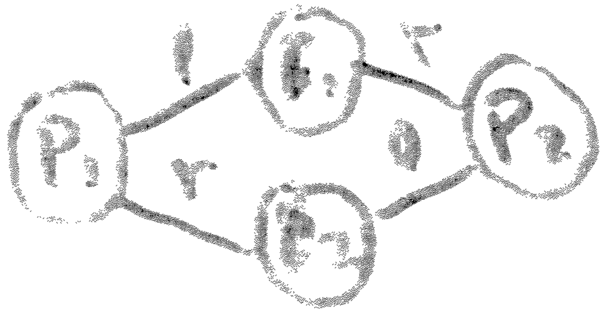
DINING PHILOSOPHERS IN ACTOR LANGUAGE

```
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 ?

```
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 = λp nil

pickup? = λm. not (eq?(m, nil))

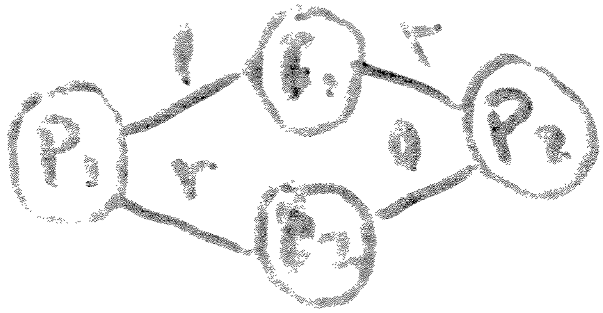
release? = λm. eq?(m, nil)

getphil = $\lambda m . m$

```
phil = rec (λb. λl. λr. λself. λc. λm.
          if (picked?(m),
              if (eq?(c, 0),
                  ready (b(l)(r)(self)(1)),
                  seq (send (l, mkrelease (self)),
                       send (r, mkrelease (self)),
                       ready (b(l)(r)(self)(2)))),
              if (released?(m),
                  if (eq?(c, 2),
                      ready( b(l)(r)(self)(1)),
                      seq (send (l, mkpickup (self)),
                           send (r, mkpickup (self)),
                           ready (b(l)(r)(self)(0)))),
              ready (b(l)(r)(self)(c)))))
```

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

mkpicked = λx. true

mkreleased = λx. false

mkpickup = λp. pr(true, p)

mkrelease = λp. pr(false, p)

pickup? = λm. if (ispr?(m), 1st(m), false)

release? = λm. if (ispr?(m), not(1st(m)), false)

picked? = λm. eq?(m, true)

released? = λm. eq?(m, false)

getphil = λm. if (ispr?(m), 2nd(m), nil)