# Chapter 4

# Actors

Distributed systems must support the addition of new components, the replacement of existing components, and dynamic changes in component interconnections. The actor model formalizes concurrent computation in distributed systems [24, 1].

The actor model assumes asynchronous communication as the most primitive interaction mechanism. In the model, the communication medium is not explicit. Actors are first-class history-sensitive entities with an explicit identity used for communication.

In response to an asynchronous message, an actor $a$ may perform one or more of the following actions:

1. send a message to an *acquaintance*, an actor whose identity is known to the actor $a$,

2. create a new actor $a'$, with a given *behavior* $b$, or

3. become *ready* to receive a new message with new behavior $b$.

The actor model theory assumes *fair* communication and computation:

- message delivery is guaranteed, and

- an actor infinitely often ready to process a message eventually processes the message.

Fairness is very useful for reasoning about equivalences of actor programs but can be hard and expensive to guarantee in practical systems when distribution, failures, and potentially complex scheduling policies must be considered.

In the following sections, we will introduce a simple actor language extending the $\lambda$ calculus, its operational semantics, a technique called *observational equivalence* to prove equivalence of actor programs, and some examples illustrating the usage of the actor language. Finally, we will refer the reader to literature describing variants and further developments of the actor model.

| | | | |
|---|---|---|---|
| $\mathcal{A}$ | $=$ | { `true`, `false`, `nil`, ... } | ***Atoms*** |
| $\mathcal{N}$ | $=$ | { 0, 1, 2, ... } | ***Natural numbers*** |
| $\mathcal{X}$ | $=$ | { $x$, $y$, $z$, ... } | ***Variable names*** |
| $\mathcal{F}$ | $=$ | { +, *, =, `ispr?`, 1st, 2nd, ... } | ***Primitive operations*** |
| | | | |
| $\mathcal{V}$ | $::=$ | | ***Values*** |
| | | $\mathcal{A} \mid \mathcal{N} \mid \mathcal{X}$ | |
| | $\mid$ | $\lambda\mathcal{X}.\mathcal{E}$ | *Functional abstraction* |
| | $\mid$ | $\mathtt{pr}(\mathcal{V},\mathcal{V})$ | *Pair constructor* |
| | | | |
| $\mathcal{E}$ | $::=$ | | ***Expressions*** |
| | | $\mathcal{V}$ | |
| | $\mid$ | $\mathcal{E}(\mathcal{E})$ | *Function application* |
| | $\mid$ | $\mathcal{F}(\mathcal{E},\ldots,\mathcal{E})$ | *Primitive function application* |
| | $\mid$ | $\mathtt{br}(\mathcal{E},\mathcal{E},\mathcal{E})$ | *Conditional execution* |
| | $\mid$ | $\mathtt{letrec}\ \mathcal{X} = \mathcal{E}\ \mathtt{in}\ \mathcal{E}$ | *Recursive definition* |
| | $\mid$ | $\mathtt{send}(\mathcal{E},\mathcal{E})$ | *Message send* |
| | $\mid$ | $\mathtt{new}(\mathcal{E})$ | *Actor creation* |
| | $\mid$ | $\mathtt{ready}(\mathcal{E})$ | *Behavior change* |

Figure 4.1: Actor Language Syntax

## 4.1 Actor Language Syntax

The actor language uses the call-by-value $\lambda$ calculus for sequential computation, and extends it with actor model primitives for coordination. An actor's behavior is modeled as a $\lambda$ calculus functional abstraction that is applied to incoming messages.

As shown in Figure 4.1, first, we extend the $\lambda$ calculus with atoms (including booleans to facilitate conditional expressions,) numbers, and primitive operators (including pair constructors and destructors to facilitate building arbitrary data structures.)[1] We then incorporate *actor primitives*: $\mathtt{send}(a, v)$ sends value $v$ to actor $a$, $\mathtt{new}(b)$ creates a new actor with behavior $b$ and returns the identity of the newly created actor, and $\mathtt{ready}(b)$ becomes ready to receive a new message with behavior $b$.

In this chapter, we use the notation $f(x)$ for function application, as opposed to the notation used in Chapter 2: $(f\ x)$. We also use standard syntactic sugar to make it easier to read and write actor programs. These defined forms are depicted in Figure 4.2.

For example, the following actor program

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

[1]Chapter 2 discussed how we can encode booleans, conditionals, numbers, and pairs in the pure $\lambda$ calculus. Sections 2.8 and 2.7 also discussed the recursion (`rec` is the same as $Y$) and sequencing combinators (`seq`) that we use here.

| | | | |
|---|---|---|---|
| $\texttt{let}\ x = e_1\ \texttt{in}\ e_2$ | $\triangleq$ | $\lambda x.e_2(e_1)$ | |
| $\texttt{seq}(e_1, e_2)$ | $\triangleq$ | $\texttt{let}\ z = e_1\ \texttt{in}\ e_2$ | $z$ *fresh* |
| $\texttt{seq}(e_1, \ldots, e_n)$ | $\triangleq$ | $\texttt{seq}(e_1, \texttt{seq}(e_2, \ldots, \texttt{seq}(e_{n-1}, e_n)) \ldots)$ | $n \geq 3$ |
| $\texttt{if}(e_1, e_2, e_3)$ | $\triangleq$ | $\texttt{br}(e_1, \lambda z.e_2, \lambda z.e_3)(\texttt{nil})$ | $z$ *fresh* |
| $\texttt{rec}(f)$ | $\triangleq$ | $\lambda x.f(\lambda y.x(x)(y))(\lambda x.f(\lambda y.x(x)(y)))$ | |

Figure 4.2: Actor Language Syntactic Sugar

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 $y$.

To create an actor with the *b5* behavior, and interact with the actor, we can write the following:

$$\texttt{send}(\texttt{new}(b5), a)$$

After executing this code, an actor with the behavior *b5* is created and eventually actor $a$ receives a message with value 5 from that newly created actor.

Another example is

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

an actor that disregards all incoming messages.

An actor does not always necessarily know its own name, as in the *b5* and *sink* examples. However, a recursive definition can be used to let an actor know its own name. For example, a *ticker* can be encoded as:

$$ticker = \texttt{rec}(\lambda b.\lambda t.\lambda n.\texttt{seq}(\texttt{send}(t, n+1), \texttt{ready}(b(t))))$$

The ticker receives messages with a natural number as its content, and in response, it sends itself a new message containing the incremented number and becomes ready to process the next message.

To create a ticker actor and get it started, we can write the following:

$$\texttt{letrec}\ t = \texttt{new}(ticker(t))\ \texttt{in}\ \texttt{send}(t, 0)$$

This actor expression creates a new ticker, initialized with its own name, and starts the ticker by sending it a message with content 0.

### 4.1.1 Join Continuations

Consider the functional programming code to compute the product of numbers in the leaves of a binary tree:

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

We could write actor code to compute the left and right branches concurrently:

```
$tprod$ = rec( $\lambda b.\lambda m.$
             seq( if( isnat?($tree(m)$),
                      send($cust(m), tree(m)$),
                      let  $newcust$ = new($joincont(cust(m)$, nil)),
                           $lp$ = new($tprod$),
                           $rp$ = new($tprod$)
                       in  seq( send($lp$, pr($left(tree(m)), newcust$)),
                                send($rp$, pr($right(tree(m)), newcust$)))),
                  ready($b$)))

$joincont$ = rec( $\lambda b.\lambda customer.\lambda firstnum.\lambda num.$
                  if( $firstnum$ = nil,
                      ready($b(customer, num)$),
                      seq( send($customer, firstnum \times num$),
                           ready($sink$))))
```

where tree product request messages and non-leaf trees are represented as pairs, and $tree = left =$ 1st, and $cust = right =$ 2nd.

In this example, a tree product actor receives a request containing a binary tree and a customer to receive the result of computing the product of the numbers in the tree. If the tree is a leaf, then the number is returned to the customer. Otherwise, two new tree product actors are created, as well as a new customer, in charge of getting the partial results from the left and right sub-trees and composing the final result for the original customer.

The new customer is also called a *join continuation*, since its behavior is to wait for two concurrent computations executing asynchronously and perform an action when they both complete. The join continuation actor has a state containing: a *customer* to be notified of completion, and a *first number* containing the result of the first computation to complete (or `nil` if none has completed.) In response to a message containing a sub-tree product, it either updates its state, saving it in its *first number* variable if it is the first computation to complete, or notifies the *customer* if it is the second and final computation to complete. Finally, it becomes a *sink* since its goals have been met.

Notice that you could create a relatively more efficient version that does not create two new actors for the left and right sub-trees, but rather uses the same tree product actor for one of the two sub-trees without sacrificing concurrency. The code would look as follows:

```
$tprod_2$ = rec( $\lambda b.\lambda self.\lambda m.$
               seq( if( isnat?($tree(m)$),
                        send($cust(m), tree(m)$),
                        letrec  $newcust$ = new($joincont(cust(m)$, nil)),
                                $rp$ = new($tprod_2(rp)$)
                         in     seq( send($self$, pr($left(tree(m)), newcust$)),
                        send(rp, pr(right(tree(m)), newcust)))),
         ready(b(self))))
```

## 4.2 Operational Semantics

The operational semantics of our actor language is defined as a set of labelled transition rules from *actor configurations* to actor configurations specifying valid computations. Concurrent systems' evolution over time can be followed by applying the rules specified in the operational semantics in a manner consistent with fairness.

### 4.2.1 Actor Configurations

Actor configurations model concurrent system components as viewed by an idealized observer, frozen in time. An actor configuration is composed of:

- a set of individually named actors, and
- messages "en-route".

An *actor configuration*, $\kappa$, denoted as:

$$\alpha \parallel \mu$$

contains an actor map, $\alpha$, which is a function mapping actor names to actor expressions, and a multi-set of messages, $\mu$. A message to actor named $a$ with content $v$ is denoted as $\langle a \Leftarrow v \rangle$.

There are two syntactic restrictions that valid actor configurations must conform to:

1. If $a \in dom(\alpha)$, then $\mathbf{fv}(\alpha(a)) \subseteq dom(\alpha)$.
2. If $\langle a \Leftarrow v \rangle \in \mu$, then $\mathbf{fv}(a) \cup \mathbf{fv}(v) \subseteq dom(\alpha)$.

The first restriction ensures that free variables in an actor expression—representing its state—refer to valid actor names in the actor configuration. The second restriction ensures that free variables in messages also refer to valid actor names in the configuration.

### 4.2.2 Reduction Contexts and Lambda Reduction Rules

An actor expression, $e$, is either a value $v$, or otherwise it can be uniquely decomposed into a *reduction context*, $\mathsf{R}$, filled with a *redex*, $\mathsf{r}$, denoted as $e = \mathsf{R} \blacktriangleright \mathsf{r} \blacktriangleleft$. The redex $\mathsf{r}$ denotes the next sub-expression to evaluate in a standard left-first, call-by-value evaluation strategy. The reduction context $\mathsf{R}$ denotes the surrounding expression with a *hole*. Figure 4.3 shows the syntax of redexes and reduction contexts for the actor language. We let $\mathsf{r}$ and $\mathsf{R}$ range over $\mathcal{E}_r$ and $\mathcal{R}$ respectively. For example, the actor expression $\mathtt{send}(\mathtt{new}(b5), a)$ can be

| | | | |
|---|---|---|---|
| $\mathcal{E}_r$ | $::=$ | | ***Redexes*** |
| | | $\mathcal{V}(\mathcal{V})$ | *Lambda application* |
| | $\mid$ | $\mathcal{F}(\mathcal{V},\ldots,\mathcal{V})$ | *Primitive function application* |
| | $\mid$ | $\texttt{br}(\mathcal{V},\mathcal{V},\mathcal{V})$ | *Conditional execution* |
| | $\mid$ | $\texttt{letrec}\ \mathcal{X}=\mathcal{V}\ \texttt{in}\ \mathcal{E}$ | *Recursive definition* |
| | $\mid$ | $\texttt{send}(\mathcal{V},\mathcal{V})$ | *Message send* |
| | $\mid$ | $\texttt{new}(\mathcal{V})$ | *Actor creation* |
| | $\mid$ | $\texttt{ready}(\mathcal{V})$ | *Behavior change* |

| | | | |
|---|---|---|---|
| $\mathcal{R}$ | $::=$ | | ***Reduction Contexts*** |
| | | $\square$ | *Hole* |
| | $\mid$ | $\mathcal{R}(\mathcal{E})$ | *Lambda application* |
| | $\mid$ | $\mathcal{V}(\mathcal{R})$ | *Lambda application* |
| | $\mid$ | $\texttt{pr}(\mathcal{R},\mathcal{E})$ | *Pair constructor* |
| | $\mid$ | $\texttt{pr}(\mathcal{V},\mathcal{R})$ | *Pair constructor* |
| | $\mid$ | $\mathcal{F}(\mathcal{V},\ldots,\mathcal{V},\mathcal{R},\mathcal{E}\ldots,\mathcal{E})$ | *Primitive function application* |
| | $\mid$ | $\texttt{br}(\mathcal{R},\mathcal{E},\mathcal{E})$ | *Conditional execution* |
| | $\mid$ | $\texttt{letrec}\ \mathcal{X}=\mathcal{R}\ \texttt{in}\ \mathcal{E}$ | *Recursive definition* |
| | $\mid$ | $\texttt{send}(\mathcal{V},\mathcal{R})$ | *Message send* |
| | $\mid$ | $\texttt{send}(\mathcal{R},\mathcal{E})$ | *Message send* |
| | $\mid$ | $\texttt{new}(\mathcal{R})$ | *Actor creation* |
| | $\mid$ | $\texttt{ready}(\mathcal{R})$ | *Behavior change* |

Figure 4.3: Redexes and Reduction Contexts

decomposed into reduction context $\texttt{send}(\square, a)$ filled with redex $\texttt{new}(b5)$. We denote this decomposition as:

$$\texttt{send}(\texttt{new}(b5), a) \quad = \quad \texttt{send}(\square, a) \blacktriangleright \texttt{new}(b5) \blacktriangleleft$$

Redexes are of two kinds: *purely functional redexes* and *actor redexes*. Figure 4.4 depicts standard $\lambda$ reduction rules for purely functional redexes. These include standard $\beta$ reduction from the $\lambda$ calculus, as well as branching, pair destructors, recursive definitions, and primitive operations.

### 4.2.3 Actor Configuration Transition Rules

The transition rules depicted in Figure 4.5 are of the form $\kappa_1 \xrightarrow{l} \kappa_2$, where $\kappa_1$ is the initial configuration, $\kappa_2$ is the final configuration, and $l$ is the transition label.[2]

There are four rules, all of which apply to an actor $a$, which we call *in focus*: the first one, labelled **fun** specifies sequential progress within the actor. The other three rules specify creation of and communication with other actors, and apply respectively to actor redexes: $\texttt{new}(b)$, $\texttt{send}(a', v)$, and $\texttt{ready}(b)$.

[2]We use $\alpha, [e]_a$ to denote the extended map $\alpha'$ which is the same as $\alpha$ except that it maps $a$ to $e$, *i.e.*, $\alpha'[a] = e \wedge \forall a' \neq a, \alpha'[a'] = \alpha[a']$. We use $\uplus$ to denote multi-set union.

$$\begin{array}{rcll}
\lambda x.e(v) & \rightarrow_\lambda & e\{v/x\} & \\
f(v_1,\ldots,v_n) & \rightarrow_\lambda & v & f \in \mathcal{F}, v = [\![f]\!](v_1,\ldots,v_n) \\
\mathtt{br}(\mathtt{true}, v, \_) & \rightarrow_\lambda & v & \\
\mathtt{br}(\mathtt{false}, \_, v) & \rightarrow_\lambda & v & \\
\mathtt{1^{st}}(\mathtt{pr}(v, \_)) & \rightarrow_\lambda & v & \\
\mathtt{2^{nd}}(\mathtt{pr}(\_, v)) & \rightarrow_\lambda & v & \\
\mathtt{letrec}\ x = v\ \mathtt{in}\ e & \rightarrow_\lambda & e\{v\{(\mathtt{letrec}\ x = v\ \mathtt{in}\ e)/x\}/x\} &
\end{array}$$

Figure 4.4: Extended $\lambda$ Calculus Reduction Rules

$$\frac{e \rightarrow_\lambda e'}{\alpha, [e]_a \parallel \mu \xrightarrow{[\mathbf{fun}:a]} \alpha, [e']_a \parallel \mu}$$

$$\alpha, [\mathsf{R} \blacktriangleright \mathtt{new}(b) \blacktriangleleft]_a \parallel \mu \xrightarrow{[\mathbf{new}:a,a']} \alpha, [\mathsf{R} \blacktriangleright a' \blacktriangleleft]_a, [\mathtt{ready}(b)]_{a'} \parallel \mu \qquad a' \textit{ fresh}$$

$$\alpha, [\mathsf{R} \blacktriangleright \mathtt{send}(a', v) \blacktriangleleft]_a \parallel \mu \xrightarrow{[\mathbf{snd}:a]} \alpha, [\mathsf{R} \blacktriangleright \mathtt{nil} \blacktriangleleft]_a \parallel \mu \uplus \{\langle a' \Leftarrow v \rangle\}$$

$$\alpha, [\mathsf{R} \blacktriangleright \mathtt{ready}(b) \blacktriangleleft]_a \parallel \{\langle a \Leftarrow v \rangle\} \uplus \mu \xrightarrow{[\mathbf{rcv}:a,v]} \alpha, [b(v)]_a \parallel \mu$$

Figure 4.5: Actor Language Operational Semantics

The **fun** rule subsumes functional computation using the extended $\lambda$ calculus reduction rules presented in Section 4.2.2.

The rule labelled **new** specifies actor creation, which applies when the focus actor $a$'s redex is $\mathtt{new}(b)$: actor $a$ creates a new actor $a'$. The behavior of $a'$ is set to the value $\mathtt{ready}(b)$, the actor $a$'s redex is replaced by the new actor's name $a'$. $a'$ must be fresh, that is, $a' \notin \mathit{dom}(\alpha) \cup \{a\}$.

The rule labelled **snd** specifies asynchronous message sending, which applies when the focus actor $a$'s redex is $\mathtt{send}(a', v)$: actor $a$ sends a message containing value $v$ to its acquaintance $a'$. Actor $a$ continues execution and the network $\mu$ is extended with a new message $\langle a' \Leftarrow v \rangle$.

The rule labelled **rcv** specifies message reception, which applies when the focus actor $a$'s redex is $\mathtt{ready}(b)$, and there is a message in $\mu$ directed to $a$, e.g., $\langle a \Leftarrow v \rangle$. The actor $a$'s new state becomes $b(v)$, that is, its behavior $b$ is applied to the incoming message value $v$. Notice that the reduction context $\mathsf{R}$ is discarded.[3]

### 4.2.4 Fairness

**Computation Sequences and Paths** If $\kappa$ is a configuration, then the *computation tree* $\tau(\kappa)$ is the set of all finite sequences of labelled transitions $[\kappa_i \xrightarrow{l_i} \kappa_{i+1} \mid i < n]$ for some $n \in \mathcal{N}$, with $\kappa = \kappa_0$. Such sequences are called

[3]See Exercise 8 for an alternative definition.

*computation sequences.* These sequences are partially ordered by the initial segment relation. A *computation path* from $\kappa$ is a maximal linearly ordered set of computation sequences in the computation tree, $\tau(\kappa)$. We denote a computation path by its maximal sequence. $\tau^{\infty}(\kappa)$ denotes the set of all (possibly infinite) paths from $\kappa$.

**Fair Computation Paths** Not all computation paths are admissible. Unfair computation paths, where enabled transitions never happen are ruled out. A transition labelled $l$ is *enabled* in a configuration $\kappa$, if and only if there is a configuration $\kappa'$ such that $\kappa \xrightarrow{l} \kappa'$.

A path $\pi = [\kappa_i \xrightarrow{l_i} \kappa_{i+1} \mid i < \infty]$ in the computation tree $\tau^{\infty}(\kappa)$ is *fair* if each enabled transition eventually happens or becomes permanently disabled. A transition with label of the form $[\mathbf{rcv} : a, v]$ becomes permanently disabled if the actor $a$ starts processing another message and never again becomes ready to accept a new message. For a configuration $\kappa$ we define $\mathcal{F}(\kappa)$ to be the subset of paths in $\tau^{\infty}(\kappa)$ that are fair. Notice that finite computation paths are fair by maximality, since all enabled transitions must have happened.

Since transition labels have sufficient information to compute computation paths, we can refer to a computation path $[\kappa_i \xrightarrow{l_i} \kappa_{i+1} \mid i < \infty]$ as $\pi = \kappa_i; l_i, l_{i+1}, \ldots$, where $\kappa_j \xrightarrow{l_j} \kappa_{j+1}$, for $j \geq i$.

## 4.3 Equivalence of Actor Programs

Programs (expressions) are considered to be *equivalent* if they behave the same when placed in any observing context.

In deterministic functional languages, such as the call-by-value $\lambda$ calculus, an observing context is a complete program with a hole, such that all the free variables in the expressions being observed are captured when the expressions are placed in the hole. "Behave the same" typically means that both converge or both diverge.

In actor expressions, we define *observing configurations* as "observing contexts", and three different notions of "behave the same" to account for non-deterministic computation.

### 4.3.1 Events, Observing Configurations, and Observations

An *observing configuration* is a configuration that contains an actor state with a hole. Instead of termination as the criterion for similar behavior, we introduce an observer primitive, `event` and observe whether or not in a given computation, `event` is executed.

We extend the operational semantics by adding the following rule:

$$\alpha, [\mathsf{R} \blacktriangleright \texttt{event}() \blacktriangleleft]_a \parallel \mu \quad \xrightarrow{[\mathbf{ev}:a]} \quad \alpha, [\mathsf{R} \blacktriangleright \texttt{nil} \blacktriangleleft]_a \parallel \mu$$

| | | | |
|---|---|---|---|
| $\mathcal{C}$ | ::= | | ***Contexts*** |
| | \| | $\square$ | *Hole* |
| | \| | $\mathcal{A} \mid \mathcal{N} \mid \mathcal{X}$ | *Value* |
| | \| | $\texttt{pr}(\mathcal{C}, \mathcal{C})$ | *Pair constructor* |
| | \| | $\lambda \mathcal{X}.\mathcal{C}$ | *Functional abstraction* |
| | \| | $\mathcal{C}(\mathcal{C})$ | *Function application* |
| | \| | $\mathcal{F}(\mathcal{C}, \ldots, \mathcal{C})$ | *Primitive function application* |
| | \| | $\texttt{br}(\mathcal{C}, \mathcal{C}, \mathcal{C})$ | *Conditional execution* |
| | \| | $\texttt{letrec}\ \mathcal{X} = \mathcal{C}\ \texttt{in}\ \mathcal{C}$ | *Recursive definition* |
| | \| | $\texttt{send}(\mathcal{C}, \mathcal{C})$ | *Message send* |
| | \| | $\texttt{new}(\mathcal{C})$ | *Actor creation* |
| | \| | $\texttt{ready}(\mathcal{C})$ | *Behavior change* |
| | \| | $\texttt{event}()$ | *Observation event* |

Figure 4.6: Contexts (Expressions with Holes) Syntax

The *observing configurations* are configurations over the extended language of the form $\alpha, [\mathsf{C}]_a \parallel \mu$, where $\mathsf{C}$ is a *hole-containing* expression or *context*. See Figure 4.6 for the syntax of contexts. We use $\mathcal{O}$ to denote the set of observing configurations, and let $\mathsf{O}$ range over $\mathcal{O}$. For a given expression $e$, the observing configurations for $e$ are those $\mathsf{O} \in \mathcal{O}$ such that filling the hole in $\mathsf{O}$ with $e$ results in a valid configuration. That is, any free variables in $e$ must refer to actor names in the configuration, i.e., $\mathbf{fv}(e) \subseteq \mathit{dom}(\alpha) \cup \{a\}$.

Since the actor language is non-deterministic, we distinguish between three different possible scenarios:

- `event` occurs in *all* possible computation paths, or
- `event` occurs in *some* computation paths, but not others, or
- `event` occurs in *no* computation paths.

We define the *observation of a computation path*, $\mathit{obs}(\pi)$, as *successful* if `event` is observed in that computation path. We then define the *observation of a configuration*, $\mathit{Obs}(\kappa)$, as a function of the observations of all its fair computation paths.

Let $\kappa$ be a configuration of the extended language, and let $\pi = [\kappa_i \xrightarrow{l_i} \kappa_{i+1} \mid i < \infty]$ be a fair path, i.e., $\pi \in \mathcal{F}(\kappa)$. Then, we define:

$$\mathit{obs}(\pi) = \begin{cases} \mathbf{s} & \text{if } (\exists i < \infty)(l_i = [\mathbf{ev} : \_]) \\ \mathbf{f} & \text{otherwise} \end{cases}$$

$$\mathit{Obs}(\kappa) = \begin{cases} \mathbf{s} & \text{if } (\forall \pi \in \mathcal{F}(\kappa))(\mathit{obs}(\pi) = \mathbf{s}) \\ \mathbf{f} & \text{if } (\forall \pi \in \mathcal{F}(\kappa))(\mathit{obs}(\pi) = \mathbf{f}) \\ \mathbf{sf} & \text{otherwise} \end{cases}$$

Consider for example the following actor expressions and observing context:

$e_1 = \texttt{send}(a, 1)$
$e_2 = \texttt{send}(a, 2)$
$e_3 = \texttt{seq}(\texttt{send}(a, 1), \texttt{send}(a, 2))$
$e_4 = \texttt{seq}(\texttt{send}(a, 2), \texttt{send}(a, 1))$

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

In the case of $\kappa_1 = \mathsf{O} \blacktriangleright e_1 \blacktriangleleft$, we observe the computation path:

$$\pi_1 = \kappa_1; [\mathbf{snd} : a'], [\mathbf{rcv} : a, 1], [\mathbf{fun} : a], \ldots, [\mathbf{fun} : a], [\mathbf{ev} : a]$$

so we say that $obs(\pi_1) = \mathbf{s}$. There are no more computation paths in the computation tree of $\kappa_1$, i.e., $\mathcal{F}(\kappa_1) = \{\pi_1\}$, because it is a deterministic computation. So we can conclude that $Obs(\kappa_1) = \mathbf{s}$.

In the case of $\kappa_2 = \mathsf{O} \blacktriangleright e_2 \blacktriangleleft$, we observe the computation path:

$$\pi_2 = \kappa_2; [\mathbf{snd} : a'], [\mathbf{rcv} : a, 2], [\mathbf{fun} : a], \ldots, [\mathbf{fun} : a]$$

in which `event` never happens, so we say that $obs(\pi_2) = \mathbf{f}$. There are no more computation paths in the computation tree of $\kappa_2$, i.e., $\mathcal{F}(\kappa_2) = \{\pi_2\}$, because it is a deterministic computation. So we can conclude that $Obs(\kappa_2) = \mathbf{f}$.

In the cases of $\kappa_3 = \mathsf{O} \blacktriangleright e_3 \blacktriangleleft$ and $\kappa_4 = \mathsf{O} \blacktriangleright e_4 \blacktriangleleft$, there are computation paths that are successful and computation paths that fail to observe the event. For example, take a computation sequence that focuses on actor $a'$ until there are no more enabled transitions on that actor. The ending configuration for both $\kappa_3$ and $\kappa_4$ would be:

$\emptyset, [\texttt{ready}(\lambda n.\texttt{if}(n = 1, \texttt{event}(), \texttt{ready}(\mathit{sink})))]_a, [\texttt{nil}]_{a'} \parallel \{\langle a \Leftarrow 1\rangle, \langle a \Leftarrow 2\rangle\}$

In this configuration there are two enabled transitions. The path that continues with the transition labeled $[\mathbf{rcv} : a, 1]$, similarly to $\pi_1$, will be successful; while the path that continues with the transition labeled $[\mathbf{rcv} : a, 2]$, similarly to $\pi_2$, will fail. So, we say $Obs(\kappa_3) = Obs(\kappa_4) = \mathbf{sf}$.

Notice that the requirement of fairness on admissible paths is critical, since otherwise we could create an observing configuration with a *ticker* actor, in which actors $a$ and $a'$ would never make progress, and therefore we would fail to observe the event in *all* computation paths.

## 4.3.2 Testing, Must, and May Equivalences

The *natural* equivalence is when equal observations are made in all observing configuration contexts. However, other two (weaker) equivalences arise if **sf** observations are considered as good as **s** observations, or if **sf** observations are considered as bad as **f** observations.

**Testing Equivalence (or convex, or Plotkin, or Egli-Milner)**

$e_1 \equiv_1 e_2$ if and only if $Obs(\mathsf{O}[e_1]) = Obs(\mathsf{O}[e_2])$ for all observing contexts $\mathsf{O} \in \mathcal{O}$.

**Must Equivalence (or upper, or Smyth)**

$e_1 \equiv_2 e_2$ if and only if $Obs(\mathsf{O}[e_1]) = \mathbf{s} \iff Obs(\mathsf{O}[e_2]) = \mathbf{s}$ for all observing contexts $\mathsf{O} \in \mathcal{O}$.

**May Equivalence (or lower, or Hoare)**

$e_1 \equiv_3 e_2$ if and only if $Obs(\mathsf{O}[e_1]) = \mathbf{f} \iff Obs(\mathsf{O}[e_2]) = \mathbf{f}$ for all observing contexts $\mathsf{O} \in \mathcal{O}$.

### 4.3.3 Equivalence Properties and Congruence

**Congruence**

By construction, testing, must and may equivalence relations are congruences:

$$e_1 \equiv_j e_2 \quad \Longrightarrow \quad \mathsf{C} \blacktriangleright e_1 \blacktriangleleft \equiv_j \mathsf{C} \blacktriangleright e_2 \blacktriangleleft \qquad \text{for } j \in \{1, 2, 3\}$$

**Partial Collapse**

While it is clear that $e_1 \equiv_1 e_2$ implies $e_1 \equiv_2 e_2 \ \wedge \ e_1 \equiv_3 e_2$, the other directions are not obvious.

Agha, Mason, Smith and Talcott [2] proved that $e_1 \equiv_2 e_2$ implies $e_1 \equiv_1 e_2$ under the assumption of fairness. This effectively collapses testing and must equivalences into one. That is, $e_1 \equiv_1 e_2 \iff e_1 \equiv_2 e_2$. Agha *et al.* also showed that $e_1 \equiv_3 e_2$ does not imply $e_1 \equiv_1 e_2$.

## 4.4 Common Examples

### 4.4.1 Reference Cell in the Actor Language

A reference cell can be encoded in the actor language as follows:

```
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)))))
```

where *get?*, *set?*, *cust*, and *contents* can be defined in terms of the actor language pairing primitives (`pr`, `ispr?`, `1st`, `2nd`.)

A client of the cell can be encoded as follows:

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

A new reference cell actor $a$ is created and three messages are sent to the cell where *mkset* and *mkget* create appropriate pairs representing *set* and *get* messages. Actor $c$ will receive a message containing either 0, 2, or 7, depending on the order of message arrival at $a$.

### 4.4.2 Mutual Exclusion in the Actor Language

Two actors can mutually exclude themselves from accessing a shared resource concurrently, i.e., they can ensure that only one of them is using the resource at a given point in time, by using a *semaphore*, as follows:

```
sem = rec( λb.λh.λm.
            if( get?(m),
                if( h = nil,
                    seq( send(cust(m), true),
                         ready(b(cust(m)))),
                    seq( send(cust(m), false),
                         ready(b(h)))),
                if( release?(m),
                    ready(b(nil)),
                    ready(b(h)))))
```

The semaphore contains state that represents who currently holds access to the shared resource, or `nil` if the resource is available. The semaphore replies `true` to the customer requesting access if allowed, and `false` otherwise.

A customer that keeps trying to get access to the shared resource may be encoded as follows:

```
customer = rec( λb.λself.λs.λm.
                 seq( if( m,
                          seq( <critical code>,
                               send(s, mkrelease())),
                          send(s, mkget(self))),
                      ready(b(self, s))))
```

Two actors $a$ and $a'$ that mutually exclude each other during the critical code sections, can be written as:

```
letrec s = new(sem(nil)),
       a = new(customer(a, s)),
       a' = new(customer(a', s)) in seq(send(a, false),
                                        send(a', false))
```

Auxiliary functions *get?*, *release?*, *mkget* and *mkrelease* can be written in terms of the actor language pairing primitives.

### 4.4.3 Dining Philosophers in the Actor Language

A typical example of the complexities of concurrent computation is the famous *dining philosophers* scenario: consider $n$ philosophers dining in a round table containing $n$ chopsticks. Each philosopher has the following sequential algorithm:

1. Pick up the *left* chopstick, if available.
2. Pick up the *right* chopstick, if available.
3. Eat.
4. Release both chopsticks.
5. Think.
6. Go to 1.

While each philosopher's algorithm seems to make sense from an individual sequential perspective, the group's concurrent behavior has the potential for *deadlock*: all philosophers can pick up their left chopsticks in step 1, and block forever in step 2, waiting for the philosopher on her/his right to release her/his left chopstick.

This well-known example highlighting the potential for deadlock in concurrent systems with shared resources, can be encoded in the actor language as follows:

```
$phil$ = rec( $\lambda b.\lambda l.\lambda r.\lambda self.\lambda sticks.\lambda m.$
            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($b(l, r, self, 0)$))))

$chopstick$ = rec( $\lambda b.\lambda h.\lambda w.\lambda m.$
                  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($b(h, w)$))))
```

A philosopher is an actor containing as internal state: its own behavior $b$; left and right chopsticks, $l$ and $r$; a reference to its own name, *self*; and the number of chopsticks it currently holds, *sticks*. On receipt of a message $m$, it checks whether it has any chopsticks. If not, it changes its internal state to now hold one chopstick. If it already has a chopstick, it is receiving the second one, thus it releases its chopsticks by sending each of them a message *mkrelease*(*self*), it attempts to pick up the left and right chopsticks again by sending them messages *mkpickup*(*self*), and it becomes ready to accept new messages with changes to its internal state to hold no chopsticks.

A chopstick is an actor containing as internal state: its own behavior $b$, a holding philosopher $h$, and a waiting philosopher $w$. On receipt of a message $m$, it checks whether it is a *pick up* message, a *release* message, or neither of these. If it is a pick up message, it checks whether a philosopher already holds the chopstick ($\mathtt{eq?}(h, \mathtt{nil})$.) If not, it sends a message to the philosopher attempting to pick it up (acknowledging it is the new chopstick holder) and changes its internal state accordingly. If a philosopher already holds the chopstick, it saves the waiting philosopher in its internal state. If it is a release message, it checks whether any philosopher is waiting for the chopstick ($\mathtt{eq?}(w, \mathtt{nil})$.) If so, it sends a message to the waiting philosopher (acknowledging it is the new chopstick holder) and changes its internal state accordingly. If not, it becomes available (and ready to be picked up.) On receipt of a different type of message (not a pick up or a release message,) the chopstick ignores it ($\mathtt{ready}(b(h, w))$.)

A table with two philosophers and two chopsticks can be modeled as follows:

$$
\begin{array}{ll}
\mathtt{letrec} & c_1 = \mathtt{new}(\mathit{chopstick}(\mathtt{nil}, \mathtt{nil})),\\
 & c_2 = \mathtt{new}(\mathit{chopstick}(\mathtt{nil}, \mathtt{nil})),\\
 & p_1 = \mathtt{new}(\mathit{phil}(c_1, c_2, p_1, 0)),\\
 & p_2 = \mathtt{new}(\mathit{phil}(c_2, c_1, p_2, 0))\ \mathtt{in} \quad e
\end{array}
$$

where $e$ is defined as:

$$
\begin{array}{ll}
e = \mathtt{seq}( & \mathtt{send}(c_1, \mathit{mkpickup}(p_1)),\\
 & \mathtt{send}(c_2, \mathit{mkpickup}(p_1)),\\
 & \mathtt{send}(c_1, \mathit{mkpickup}(p_2)),\\
 & \mathtt{send}(c_2, \mathit{mkpickup}(p_2)))
\end{array}
$$

To complete this example, we need to provide the auxiliary definitions:

$$
\begin{array}{ll}
\mathit{mkpickup} & = \lambda p.\mathtt{pr}(\mathtt{true}, p)\\
\mathit{mkrelease} & = \lambda p.\mathtt{pr}(\mathtt{false}, p)\\
\mathit{pickup?} & = \lambda m.\mathtt{if}(\mathtt{ispr?}(m), \mathtt{1}^{\mathtt{st}}(m), \mathtt{false})\\
\mathit{release?} & = \lambda m.\mathtt{if}(\mathtt{ispr?}(m), \mathtt{not}(\mathtt{1}^{\mathtt{st}}(m)), \mathtt{false})\\
\mathit{getphil} & = \lambda m.\mathtt{if}(\mathtt{ispr?}(m), \mathtt{2}^{\mathtt{nd}}(m), \mathtt{nil})
\end{array}
$$

## 4.5 Bibliographic Notes

The $\pi$ calculus presented in Chapter 3 as well as other process algebras (such as CCS and CSP) take synchronous communication as the most primitive form of communication. In contrast, the actor model assumes asynchronous communication as the most primitive. In the $\pi$ calculus, channels explicitly model the inter-process communication medium. Multiple processes can share a channel, potentially causing unexpected interference, whereby two processes can be listening on the same channel. In contrast, actors have unique identities used for communication which can still be passed around to dynamically change the communication topology, but do not cause potential interference since all messages directed to an actor eventually reach it.

Chapter 4 followed closely the actor language syntax, semantics, and several examples from the thorough presentation by Agha, Mason, Smith and Talcott [2]. Following are key differences for the interested reader.

Agha *et al.*'s language uses `become` as an actor primitive rather than our `ready` primitive. The key difference is that `become` creates a new anonymous actor to carry out the rest of the computation (represented by the reduction context surrounding the `become` primitive) and makes the actor in focus immediately ready to receive new messages. In contrast, our `ready` semantics discards the rest of the computation (it assumes that code after `ready` is *dead*, which is not an issue if `ready` always appears in tail form position.) The advantage of our semantics is that it resembles more closely practical actor language implementations, where new actor creation is expensive and should be avoided if unnecessary. It also resembles more closely actor languages that extend object-oriented programming languages, where messages are modeled as potential method invocations (see Chapter 9.)

Agha *et al.*'s language uses `letactor` as syntactic sugar over actor primitives `newadr` and `initbeh` for actor creation. In our presentation, we chose to use a `new` primitive actor expression that creates an actor and returns its (fresh) address. We combine it with a `letrec` construct to enable an actor to be initialized with its own address. Once again, the advantage of `new` over `letactor/newadr/initbeh` is that it is easier to understand the model, and it more closely resembles practical language implementations where (extended and modified) objects are used to model actors.

Agha *et al.*'s presentation discusses *composability* of actor configurations. To this extent, it presents configurations as explicitly including an interface with their surrounding environment, modeled by a set of *receptionists*, and a set of *external actors*. However, the composition of two configurations is only allowed if their internal actor names do not collapse, that is, there is no operation equivalent to $\alpha$-renaming in the $\lambda$ or $\pi$ calculi. In part, this is due to the non-local effect of such an operation (an actor name may have been *exported* to other configurations,) however, it should be possible for *non-receptionist* actors to perform $\alpha$-renaming. For the purpose of simplicity and ease of understanding, we chose not to include composability in this chapter's presentation, and rather defer its presentation to Section 7.3.2. Finally, Agha *et al.* present basic expression equivalence laws for actor primitives, as well as several techniques for proving actor expression equivalence. Readers interested in these equational laws and proof techniques are referred to [2].

From a programming language perspective, actors can be thought of as *active objects*, reactive entities that process messages sequentially from a *mailbox* buffering asynchronous messages (which can be modeled as potential method invocations.) From an artificial intelligence perspective, actors can be thought of as the core of *software agents*. Actors model the most elemental computation and communication capabilities of software agents. Additional agent capabilities such as higher-level coordination, planning, and knowledge, must be explicitly formalized in higher-level models.

Several works have extended the actor model presented in this chapter to formalize higher-level aspects of distributed computing. Following are some examples for the interested reader. Field and Varela created the $\tau$-calculus to model globally consistent distributed state in the face of failures [19]. Varela and Agha created abstractions for grouping actors into *casts* managed by *directors* to accomplish hierarchical coordination [52]. Toll and Varela introduced primitives for mobility and security to model authentication and access control in open environments [47]. Morali and Varela incorporated trust management primitives to model trust propagation in electronic commerce systems [37]. Jamali and Agha defined a *cyber-organism* abstraction to model shared and constrained resources [26]. Ren and Agha explored real-time synchronization primitives to model soft real-time computing [43].

The actor model has influenced the development of several programming languages. Sussman and Steele developed Scheme in 1975 in an attempt to understand the actor model first conceived by Carl Hewitt [46]. Early actor languages developed at MIT include PLASMA (1975) and Act1 (1981). More recent actor languages include Acore (1987), Rossette (1989), ABCL (1990), Erlang (1993), Obliq (1994), E (1998) and SALSA (1999).

## 4.6 Exercises

1. Write a functional expression that computes *fib(n)*. Then, write a behavior for an actor that computes *fib(n)* concurrently using a join continuation.

2. Consider the reference cell example in Section 4.4.1. Write *get?*, *set?*, *cust*, *contents*, *mkset* and *mkget* using the $\lambda$ calculus extended with pairing primitives defined in Figure 4.1.

3. Modify the reference cell example in Section 4.4.1 to notify a customer when the cell value is updated (such as is done in the $\pi$ calculus example in Section 3.4.1.)

4. Consider the dining philosophers example in Section 4.4.3. An alternative definition of auxiliary functions *mkpickup* and *mkrelease* follows:

   $$\begin{aligned} \mathit{mkpickup} &= \lambda p.p \\ \mathit{mkrelease} &= \lambda p.\texttt{nil} \end{aligned}$$

   Define the remaining auxiliary functions using this alternative definition:

   $$\begin{aligned} \mathit{pickup?} &= \ldots \\ \mathit{release?} &= \ldots \\ \mathit{getphil} &= \ldots \end{aligned}$$

5. Modify the dining philosophers example in Section 4.4.3 so that philosophers use a *semaphore* to access the chopsticks. You should reuse the *semaphore* example in Section 4.4.2. Discuss the advantages and disadvantages of each implementation.

6. Considering the *b5* behavior example in Section 4.1 and the following actor configuration:
$$\kappa = \emptyset, [\mathtt{send}(\mathtt{new}(b5), a)]_{a'} \ \| \ \emptyset$$
use the operational semantics rules to evolve the configuration to a final state, where no further transitions are enabled.

7. What problems may arise if an actor configuration does not follow the syntactic restrictions described in Section 4.2.1? Does the model prevent ill-formed messages (e.g., a message directed to a number) from appearing in actor configurations? What about ill-formed behaviors (e.g., non-lambda abstractions)?

8. Modify the actor semantics so that on message reception, the reduction context of the $\mathtt{ready}(b)$ redex is not discarded. Instead, a new anonymous actor should be created to carry out the remaining computation.[4] This new anonymous actor can create other actors and send messages, but it cannot receive new messages, since its name is unknown. (Hint: you need to modify the **rcv** rule.)

9. Given the actor expression $\mathtt{seq}(\mathtt{send}(a, 2), \mathtt{send}(a, 1))$, eliminate syntactic sugar (`seq`) and decompose the expression into a reduction context and a redex.

10. Prove that all actor expressions are uniquely decomposable into a reduction context and a redex.

11. Consider the following two expressions:
$e_1 = \mathtt{ready}(\lambda n.\mathtt{seq}(\mathtt{send}(a, \mathtt{nil}), \mathtt{ready}(sink)))$
$e_2 = \mathtt{ready}(\lambda n.\mathtt{if}(n = 1, \mathtt{send}(a, \mathtt{nil}), \mathtt{ready}(sink)))$

    (a) Create an observing context that can **not** distinguish between $e_1$ and $e_2$.

    (b) Create an observing context that can distinguish between $e_1$ and $e_2$.

12. Prove that $e_1 \equiv_3 e_2$ does not imply $e_1 \equiv_1 e_2$.

[4]This is the semantics of the `become` primitive in [2].