Actor Model

Open 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 open distributed systems (Hewitt, 1977; Agha, 1986).
Actor Model–Asynchronous Communication

- The actor model assumes *asynchronous* communication to be 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 $b$. 
Concurrent programming with actors

In response to a message, an actor can:

1. modify its local state,
2. create new actors, and/or
3. send messages to acquaintances.
Actor Model–Open Systems

Openness in the actor model is thus supported as follows:

- new components can be added by creating new actors dynamically,
- replacement of existing components is modeled by modifying an actor’s behavior, and
- dynamic changes in component interconnections are modeled by passing actor identities in messages and subsequently modifying actor behaviors, thereby altering the actor reference graph and potential future communications.
Actor Model–Fairness

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.
Agha, Mason, Smith, and Talcott’s Language

We will introduce a simple actor language (AMST):

- extending the $\lambda$ calculus, with pairs, conditionals, and actor primitives
- defining its operational semantics, as transitions between actor configurations
- illustrating a technique called *observational equivalence* to prove equivalence of actor programs, and
- studying examples illustrating the usage of the actor language.
\textbf{\lambda Calculus Syntax and Semantics}

The \textit{syntax} for \lambda calculus expressions is

\[ e ::= v \quad \text{– variable} \]
\[ | \quad \lambda v. e \quad \text{– functional abstraction} \]
\[ | \quad (e \ e) \quad \text{– function application} \]

The \textit{semantics} is essentially defined by \textit{\beta-reduction}:

\[ (\lambda x. E \ M) \xrightarrow{\beta} E\{M/x\}. \]

For example, given the expression \((\lambda x. x \ y)\):

\[ (\lambda x. \underbrace{x}_{E} \underbrace{y}_{M}), \]  

We replace free \(x\)’s in \(E\) with \(M\), to obtain: \((\lambda x.x \ y) \xrightarrow{\beta} y\).
Actor Language Syntax-I

\[ \mathcal{A} = \{ \text{true, false, nil, \ldots} \} \] \hspace{2cm} \text{Atoms}

\[ \mathcal{N} = \{ 0, 1, 2, \ldots \} \] \hspace{2cm} \text{Natural numbers}

\[ \mathcal{X} = \{ x, y, z, \ldots \} \] \hspace{2cm} \text{Variable names}

\[ \mathcal{F} = \{ +, \times, =, \text{ispr?}, 1^{\text{st}}, 2^{\text{nd}}, \ldots \} \] \hspace{2cm} \text{Primitive operators}

\[ \mathcal{V} ::= \]

\[ \mathcal{A} | \mathcal{N} | \mathcal{X} \] \hspace{2cm} \text{Values}

\[ | \lambda \mathcal{X}.\mathcal{E} \] \hspace{2cm} \text{Functional abstraction}

\[ | \text{pr}(\mathcal{V}, \mathcal{V}) \] \hspace{2cm} \text{Pair constructor}
Actor Language Syntax-II

\[ \mathcal{E} ::= \]

\[ \forall \]

\[ \quad \text{Pair constructor} \]

\[ \text{pr}(\mathcal{E}, \mathcal{E}) \]

\[ \text{Function application} \]

\[ \mathcal{E}(\mathcal{E}) \]

\[ \text{Primitive function application} \]

\[ \mathcal{F}(\mathcal{E}, \ldots, \mathcal{E}) \]

\[ \text{Conditional execution} \]

\[ \text{br}(\mathcal{E}, \mathcal{E}, \mathcal{E}) \]

\[ \text{Recursive definition} \]

\[ \text{letrec } X = \mathcal{E} \text{ in } \mathcal{E} \]

\[ \text{Message send} \]

\[ \text{send}(\mathcal{E}, \mathcal{E}) \]

\[ \text{Actor creation} \]

\[ \text{new}(\mathcal{E}) \]

\[ \text{Behavior change} \]

\[ \text{ready}(\mathcal{E}) \]
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.

• 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, e.g., $1^{st}(\text{pr}(x, y)) = x$ and $2^{nd}(\text{pr}(x, y)) = y$. 
Actor Language Syntax

We then incorporate *actor primitives*:

- **send**(a, v) sends value v to actor a,
- **new**(b) creates a new actor with behavior b and returns the identity of the newly created actor, and
- **ready**(b) becomes ready to receive a new message with behavior b.
Actor Language Syntactic Sugar

We also define some syntactic sugar for definitions, sequencing, conditionals and recursive functions which get translated to the $\lambda$ calculus as follows:

$\text{let } x = e_1 \text{ in } e_2 \equiv \lambda x. e_2(e_1)$

$\text{seq}(e_1, e_2) \equiv \text{let } z = e_1 \text{ in } e_2 \quad z \text{ fresh}$

$\text{seq}(e_1, \ldots, e_n) \equiv \text{seq}(e_1, \text{seq}(e_2, \ldots, \text{seq}(e_{n-1}, e_n)) \ldots) \quad n \geq 3$

$\text{if}(e_1, e_2, e_3) \equiv \text{br}(e_1, \lambda z. e_2, \lambda z. e_3)(\text{nil}) \quad z \text{ fresh}$

$\text{rec}(f) \equiv \lambda x. f(\lambda y. x(x)(y))(\lambda x. f(\lambda y. x(x)(y)))$
AMST Actor Language Example 1

The following actor program

\[ b5 = \text{rec}(\lambda y.\lambda x.\text{seq}(\text{send}(x, 5), \text{ready}(y))) \]

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 \( y \) (\( b5 \)).

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

\[ \text{send(new}(b5), a) \]

When 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.
AMST Actor Language Example 2

Another example is

\[ \text{sink} = \text{rec}(\lambda b. \lambda m. \text{ready}(b)) \]

a behavior for an actor that disregards all incoming messages.
AMST Actor Language Example 3

A recursive definition can be used to let an actor know its own name. For example, a *ticker* can be encoded as:

\[
ticker = \text{rec}(\lambda b.\lambda t.\lambda n.\lambda m.\text{seq}(\text{send}(t, \text{nil}), \text{ready}(b(t)(n + 1))))
\]

The ticker has as its state

- an actor name \( (t) \) and
- a natural number \( (n) \),

and in response to (tick) messages \( (m) \),

- it sends the actor a new message and
- becomes ready to process the next message with an incremented internal time.
AMST Actor Language Example 3

A recursive definition can be used to let an actor know its own name. For example, a *ticker* can be encoded as:

\[
ticker = \text{rec}(\lambda b.\lambda t.\lambda n.\lambda m.\text{seq}(\text{send}(t, \text{nil}), \text{ready}(b(t)(n + 1))))
\]

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

\[
\text{letrec } t = \text{new}(ticker(t)(0)) \text{ in send}(t, \text{nil})
\]

This actor expression creates a new ticker, initialized with its own name and time 0, and starts the ticker by sending it a tick message, represented as `nil`. 
Reference Cell in the AMST Actor Language

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

\[
cell = \text{rec}( \lambda b. \lambda c. \lambda m. \\
\quad \text{if}(\text{get?}(m), \\
\quad \quad \text{seq}(\text{send}(\text{cust}(m), c), \\
\quad \quad \quad \text{ready}(b(c))), \\
\quad \quad \text{if}(\text{set?}(m), \\
\quad \quad \quad \text{ready}(b(\text{contents}(m))), \\
\quad \quad \quad \text{ready}(b(c))))))
\]

A client of the cell can be encoded as follows:

\[
\text{let } a = \text{new}(\text{cell}(0)) \text{ in seq}(\text{send}(a, \text{mkset}(7)), \\
\quad \text{send}(a, \text{mkset}(2)), \\
\quad \text{send}(a, \text{mkget}(c)))
\]
Reference Cell in the AMST Actor Language

Exercises

1. What value will actor $c$ receive after the example is executed?

2. How would you define the auxiliary $\lambda$ calculus functions:
   
   \[
   \begin{align*}
   mkget &= \quad = \\
   mkset &= \quad = \\
   get? &= \quad = \\
   set? &= \quad = \\
   cust &= \quad = \\
   contents &= \quad = \\
   \end{align*}
   \]

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