

Concurrent Programming with Actors (PDCS 4, PDCS 9, CPE 5*)

Support for the actor model in SALSA and Erlang

Carlos Varela
Rennselaer Polytechnic Institute

October 6, 2026

* Concurrent Programming in Erlang, by J. Armstrong, R. Virding, C. Wikström, M. Williams

Agha, Mason, Smith & Talcott

1. Extend a functional language (call-by-value λ calculus + `ifs` and `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.

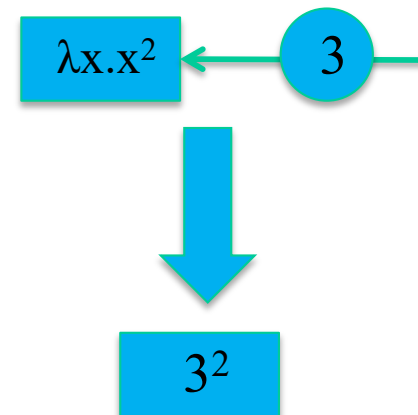
λ -Calculus as a Model for Sequential Computation

Syntax:

e	$::=$	v	<i>variable</i>
	$ $	$\lambda v . e$	<i>function</i>
	$ $	$e (e)$	<i>application</i>

Example of beta-reduction:

$$\lambda x . x^2 (3) \longrightarrow x^2 \{ 3 / x \}$$



Actor Primitives

- `send (a, v)`
 - Sends value v to actor a .
- `new (b)`
 - Creates a new actor with behavior b (a λ -calculus functional abstraction) and returns the identity/name of the newly created actor.
- `ready (b)`
 - Becomes ready to receive a new message with behavior b .

AMST Actor Language

Examples

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

Sample usage:

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

A *sink*, an actor that disregards all messages:

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

Operational Semantics for AMST Actor Language

- Operational semantics of actor model as a labeled transition relationship between actor configurations:

$$k_1 \xrightarrow{[\text{label}]} k_2$$

- Actor configurations model open system components:
 - Set of individually named actors
 - Messages “en-route”

Actor Configurations

$$\mathbf{k} = \alpha \parallel \mu$$

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

μ is a multi-set of messages “en-route.”

Syntactic restrictions on configurations

Given $A = \text{Dom}(\alpha)$:

- If a in A , then $\text{fv}(\alpha(a))$ is a subset of A .
- If $\langle a \leq v \rangle$ in μ , then $\{a\} \cup \text{fv}(v)$ is a subset of A .

Reduction contexts and redexes

Consider the expression:

$e = \text{send}(\text{new}(b5), a)$

- The redex r represents the next sub-expression to evaluate in a left-first call-by-value evaluation strategy.
- The reduction context R (or *continuation*) is represented as the surrounding expression with a *hole* replacing the redex.

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

$e = R \blacktriangleright r \blacktriangleleft$ where

$R = \text{send}(\square, a)$

$r = \text{new}(b5)$

Operational Semantics of Actors

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

$$\alpha, [R \blacktriangleright \text{new}(b) \blacktriangleleft]_a \parallel \mu \xrightarrow{[\text{new}:a,a']} \alpha, [R \blacktriangleright a' \blacktriangleleft]_a, [\text{ready}(b)]_{a'} \parallel \mu$$

a' fresh

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

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

Operational semantics example (1)

$$k_0 = [\text{send}(\square, a) \blacktriangleright \text{new}(b5) \blacktriangleleft]_a \parallel \{\}$$

$$k_1 = [\text{send}(b, a)]_a, [\text{ready}(b5)]_b \parallel \{\}$$

$$k_0 \xrightarrow{[\text{new}: a, b]} k_1$$

$$k_2 = [\text{nil}]_a, [\text{ready}(b5)]_b \parallel \{< b \leq a >\}$$

$$k_1 \xrightarrow{[\text{snd}: a]} k_2$$

Operational semantics example (2)

$k_2 = [\text{nil}]_a, [\text{ready}(b5)]_b \parallel \{ \langle b \leq a \rangle \}$

$k_3 = [\text{nil}]_a,$

$[\text{rec}(\lambda y. \lambda x. \text{seq}(\text{send}(x, 5), \text{ready}(y))) (a)]_b$

$\parallel \{ \}$

$k_2 \xrightarrow{[\mathbf{rcv}:b,a]} k_3$

$k_4 = [\text{nil}]_a, [\text{seq}(\text{send}(a, 5), \text{ready}(b5))]_b$

$\parallel \{ \}$

$k_3 \xrightarrow{[\mathbf{fun}:b]} k_4$

Operational semantics example (3)

$$k_4 = [nil]_a, \\ [seq(\square, ready(b5)) \blacktriangleright send(a, 5) \blacktriangleleft]_b \\ || \quad \{\}$$

$$k_4 \xrightarrow{[snd:b]} k_5$$

$$k_5 = [nil]_a, [seq(nil, ready(b5))]_b \\ || \quad \{< a \leq 5 >\}$$

Operational semantics example (4)

$$k_5 = [nil]_a, [seq(nil, ready(b5))]_b \\ \parallel \{ \langle a \leq 5 \rangle \}$$
$$k_6 = [nil]_a, [ready(b5)]_b \parallel \{ \langle a \leq 5 \rangle \}$$
$$k_5 \xrightarrow{[fun:b]} k_6$$

Semantics example summary

$k_0 = [\text{send}(\text{new}(b5), a)]_a \parallel \{\}$

$k_6 = [\text{nil}]_a, [\text{ready}(b5)]_b \parallel \{< a \leq 5 >\}$

$k_0 \xrightarrow{[\text{new}: a, b]} k_1 \xrightarrow{[\text{snd}: a]} k_2 \xrightarrow{[\text{rcv}: b, a]} k_3 \xrightarrow{[\text{fun}: b]} k_4$

$k_4 \xrightarrow{[\text{snd}: b]} k_5 \xrightarrow{[\text{fun}: b]} k_6$

This sequence of (labeled) transitions from k_0 to k_6 is called a *computation sequence*.

Reference Cell

```
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)) )
```

Asynchronous communication

$$k_0 = [\text{ready}(\text{cell}(0))]_a \\ \parallel \{ \langle a \leq s(7) \rangle, \langle a \leq s(2) \rangle, \langle a \leq g(c) \rangle \}$$

Three receive transitions are enabled at k_0 .

Multiple enabled transitions can lead to *nondeterministic* behavior

$$k_0 \xrightarrow{[\text{rcv}: a, s(7)]} k_1$$

$$k_0 \xrightarrow{[\text{rcv}: a, s(2)]} k_1'$$

$$k_0 \xrightarrow{[\text{rcv}: a, g(c)]} k_1''$$

The set of all computations sequences from k_0 is called the *computation tree* $\tau(k_0)$.

Nondeterministic behavior (1)

$k_0 = [\text{ready}(\text{cell}(0))]_a$
 $\parallel \{ \langle a \leq s(7) \rangle, \langle a \leq s(2) \rangle, \langle a \leq g(c) \rangle \}$

$k_1 \xrightarrow{*} [\text{ready}(\text{cell}(7))]_a$
 $\parallel \{ \langle a \leq s(2) \rangle, \langle a \leq g(c) \rangle \}$

Customer c will get 2 or 7.

$k_1' \xrightarrow{*} [\text{ready}(\text{cell}(2))]_a$
 $\parallel \{ \langle a \leq s(7) \rangle, \langle a \leq g(c) \rangle \}$

$k_1'' \xrightarrow{*} [\text{ready}(\text{cell}(0))]_a$
 $\parallel \{ \langle a \leq s(7) \rangle, \langle a \leq s(2) \rangle, \langle c \leq 0 \rangle \}$

Customer c will get 0.

Nondeterministic behavior (2)

$$k_0 = [\text{ready}(\text{cell}(0))]_a \parallel \{ \langle a \leq s(7) \rangle, \langle a \leq s(2) \rangle, \langle a \leq g(c) \rangle \}$$

Order of three receive transitions determines final state, e.g.:

$$k_0 \xrightarrow{[\mathbf{rcv}: a, g(c)]} k_1 \xrightarrow{*} \xrightarrow{[\mathbf{rcv}: a, s(7)]} k_2 \xrightarrow{*} \xrightarrow{[\mathbf{rcv}: a, s(2)]} k_3$$

$$k_f = [\text{ready}(\text{cell}(2))]_a \parallel \{ \langle c \leq 0 \rangle \}$$

Final cell state is 2.

Nondeterministic behavior (3)

$$k_0 = [\text{ready}(\text{cell}(0))]_a \parallel \{ \langle a \leq s(7) \rangle, \langle a \leq s(2) \rangle, \langle a \leq g(c) \rangle \}$$

Order of three receive transitions determines final state, e.g.:

$$k_0 \xrightarrow{[\mathbf{rcv}: a, s(2)]} k_1 \xrightarrow{*} \xrightarrow{[\mathbf{rcv}: a, g(c)]} k_2 \xrightarrow{*} \xrightarrow{[\mathbf{rcv}: a, s(7)]} k_3$$

$$k_f = [\text{ready}(\text{cell}(7))]_a \parallel \{ \langle c \leq 2 \rangle \}$$

Final cell state is 7.

Erlang support for Actors

- Actors in Erlang are modeled as *processes*. Processes start by executing an arbitrary *function*. Related functions are grouped into *modules*.
- Messages can be any Erlang *terms*, *e.g.*, atoms, tuples (fixed arity), or lists (variable arity). Messages are sent asynchronously.
- State is modeled implicitly with function arguments. Actors explicitly call `receive` to get a message, and must use tail-recursion to get new messages, *i.e.*, control loop is explicit.

Reference Cell in Erlang

```
-module(cell) .  
-export([cell/1]) .  
  
cell(Content) ->  
    receive  
        {set, NewContent} -> cell(NewContent) ;  
        {get, Customer}    -> Customer ! Content,  
                               cell(Content)  
    end.
```

Reference Cell in Erlang

```
-module(cell) .  
-export([cell/1]) .
```

```
cell(Content) ->
```

Encapsulated state Content.

```
    receive
```

Message
handlers

```
    {set, NewContent} -> cell(NewContent) ;  
    {get, Customer}   -> Customer ! Content,  
                        cell(Content)
```

State change.

```
end.
```

Explicit control loop: Actions at the end of a message need to include tail-recursive function call. Otherwise actor (process) terminates.

Reference Cell in Erlang

```
-module(cell) .
```

```
-export([cell/1]) .
```

```
cell(Content) ->
```

```
    receive
```

```
        {set, NewContent} -> cell(NewContent) ;
```

```
        {get, Customer}    -> Customer ! Content,  
                               cell(Content)
```

```
    end.
```

Content is an argument to the cell function.

{set, NewContent} is a tuple *pattern*. set is an atom. NewContent is a variable.

Messages are checked one by one, and for each message, first pattern that applies gets its actions (after ->) executed. If no pattern matches, messages remain in actor's mailbox.

Cell Tester in Erlang

```
-module(cellTester) .  
-export([main/0]) .  
  
main() -> C = spawn(cell,cell,[0]) ,  
          C!{set,7} ,  
          C!{set,2} ,  
          C!{get,self()} ,  
          receive  
              Value ->  
                  io:format("~w~n",[Value])  
          end.
```

Cell Tester in Erlang

```
-module(cellTester) .
```

```
-export([main/0]) .
```

```
main() -> C = spawn(cell,cell,[0]),  
          C!{set,7},  
          C!{set,2},  
          C!{get,self()},  
          receive  
              Value ->  
                  io:format("~w~n",[Value])  
          end.
```

Actor creation (spawn)

Message passing (!)

receive waits until a message is available.

Cell Tester in Erlang

```
-module(cellTester) .  
-export([main/0]) .  
  
main() -> C = spawn(cell,cell,[0]) ,  
          C!{set,7} ,  
          C!{set,2} ,  
          C!{get,self()} ,  
          receive  
              Value ->  
                  io:format("~w~n",[Value])  
          end.
```

[0] is a *list* with the arguments to the module's function. General form:

`spawn(module, function, arguments)`

Function calls take the form:
`module:function(args)`

`self()` is a *built-in function (BIF)* that returns the process id of the current process.

Actors/SALSA

- Actor Model

- A reasoning framework to model concurrent computations
- Programming abstractions for distributed open systems

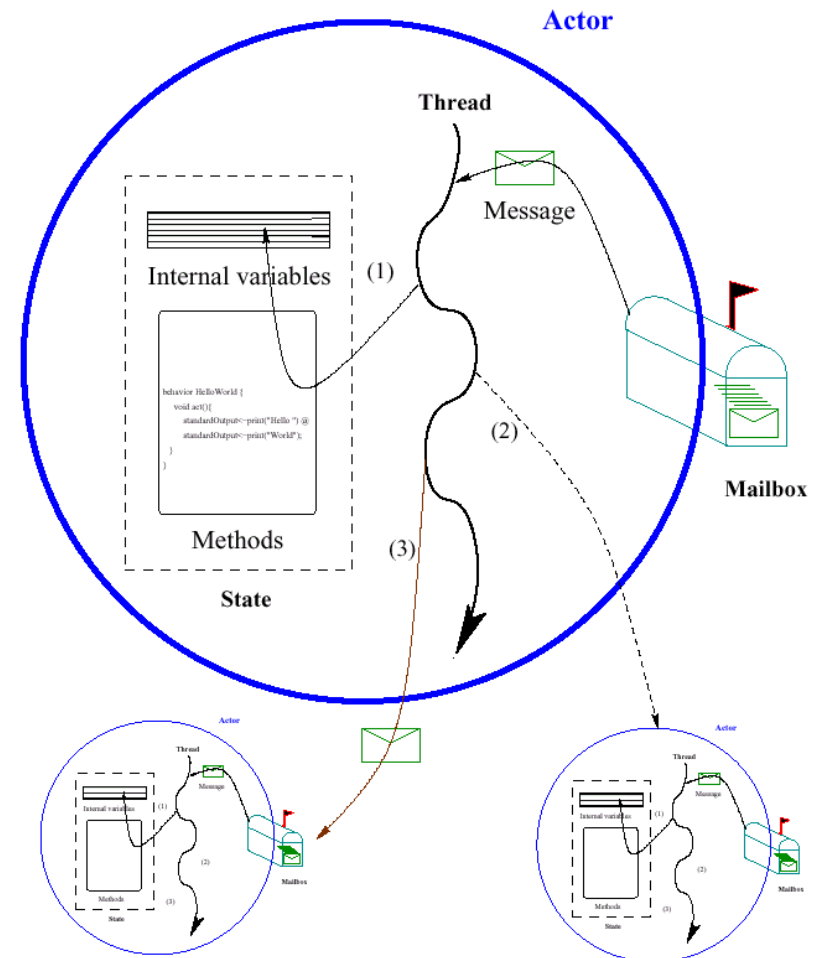
G. Agha, *Actors: A Model of Concurrent Computation in Distributed Systems*. MIT Press, 1986.

Agha, Mason, Smith and Talcott, “A Foundation for Actor Computation”, *J. of Functional Programming*, 7, 1-72, 1997.

- SALSA

- Simple Actor Language System and Architecture
- An actor-oriented language for mobile and internet computing
- Programming abstractions for internet-based concurrency, distribution, mobility, and coordination

C. Varela and G. Agha, “Programming dynamically reconfigurable open systems with SALSA”, *ACM SIGPLAN Notices, OOPSLA 2001*, 36(12), pp 20-34.



SALSA support for Actors

- Programmers define *behaviors* for actors. Actors are instances of behaviors.
- Messages are modeled as potential method invocations. Messages are sent asynchronously.
- State is modeled as encapsulated objects/primitive types.
- Tokens represent future message return values. Continuation primitives are used for coordination.

Reference Cell Example

```
module cell;

behavior Cell {
    Object content;

    Cell(Object initialContent) {
        content = initialContent;
    }

    Object get() { return content; }

    void set(Object newContent) {
        content = newContent;
    }
}
```

Reference Cell Example

```
module cell;
```

```
behavior Cell {  
    Object content;
```

Encapsulated state content.

```
    Cell(Object initialContent) {  
        content = initialContent;  
    }
```

Actor constructor.

```
    Object get() { return content; }
```

```
    void set(Object newContent) {  
        content = newContent;  
    }
```

Message handlers.

State change.

```
}
```

Reference Cell Example

```
module cell;

behavior Cell {
    Object content;

    Cell(Object initialContent) {
        content = initialContent;
    }

    Object get() { return content; }

    void set(Object newContent) {
        content = newContent;
    }
}
```

return asynchronously
sets token associated
to get message.

Implicit control loop:
End of message implies
ready to receive next
message.

Cell Tester Example

```
module cell;

behavior CellTester {

    void act( String[] args ) {

        Cell c = new Cell(0);
        c <- set(7);
        c <- set(2);
        token t = c <- get();
        standardOutput <- println( t );
    }
}
```

Cell Tester Example

```
module cell;

behavior CellTester {

    void act( String[] args ) {

        Cell c = new Cell(0);
        c <- set(7);
        c <- set(2);
        token t = c <- get();
        standardOutput <- println( t );

    }
}
```

Actor creation (new)

Message passing (<-)

println message can
only be processed when
token t from c's
get() message handler
has been produced.

Cell Tester Example

```
module cell;

behavior CellTester {

    void act( String[] args ) {

        Cell c = new Cell(0);
        c <- set(7);
        c <- set(2);
        token t = c <- get();
        standardOutput <- println( t );

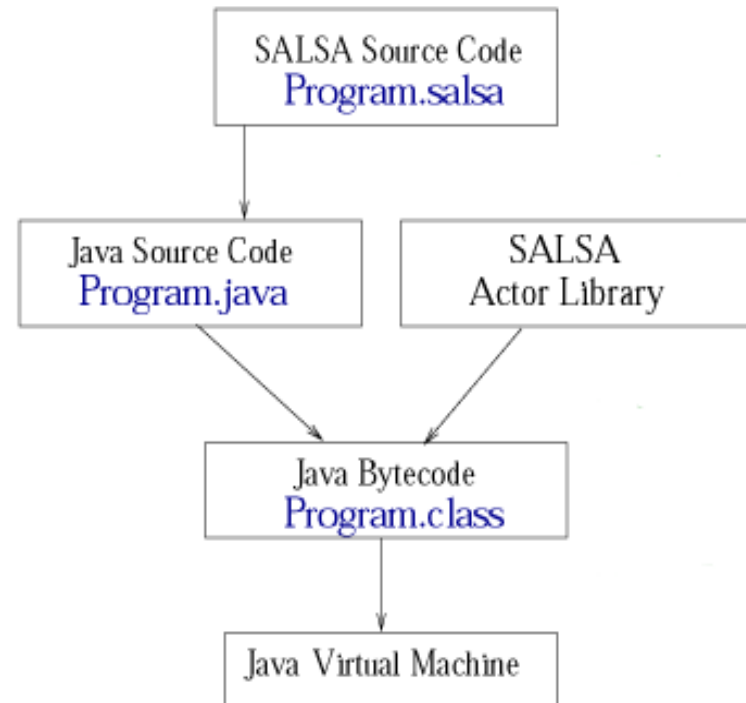
    }

}
```

All message passing is asynchronous.

println message is called *partial* until *token* *t* is produced. Only *full* messages (with no pending tokens) are delivered to actors.

SALSA compiles to Java



- SALSA source files are compiled into Java source files before being compiled into Java byte code.
- SALSA programs may take full advantage of the Java API.

Summary

- Actors are concurrent entities that react to messages.
 - State is completely encapsulated. There is no shared memory!
 - Message passing is asynchronous.
 - Actor run-time has to ensure fairness.
- AMST extends the call by value lambda calculus with actor primitives. State is modeled as function arguments. Actors use `ready` to receive new messages.
- Erlang extends a functional programming language core with processes that run arbitrary functions. State is implicit in the function's arguments. Control loop is explicit: actors use `receive` to get a message, and tail-form recursive call to continue.
- SALSA extends an object-oriented programming language (Java) with universal actors. State is encapsulated in instance variables. Control loop is implicit: ending a message handler, signals readiness to receive a new message.

Exercises

41. Define pairing primitives (`pr`, `1st`, `2nd`) in the pure lambda calculus.
42. Download and execute the reference cell examples in SALSA and Erlang.
43. PDCS Exercise 9.6.1 (page 203). Modify the behavior to reuse the tree product actor to compute the product of the left subtree. (See PDCS page 63 for the corresponding `tprod2` behavior in AMST.)
44. PDCS Exercise 9.6.4 (page 204).