Concurrent Programming with Actors
(PDCS 9, CPE 5*)
Support for the actor model in SALSA and Erlang

Carlos Varela
Rensselaer Polytechnic Institute

October 10, 2023

* Concurrent Programming in Erlang, by J. Armstrong, R. Virding, C. Wikström, M. Williams
1. Extend a functional language (call-by-value λ calculus + if statements 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.
\[ \lambda \text{-Calculus as a Model for Sequential Computation} \]

Syntax:

\[
\begin{align*}
e & ::= v & \text{variable} \\
& | \ \lambda v. e & \text{function} \\
& | e(e) & \text{application}
\end{align*}
\]

Example of beta-reduction:

\[
\lambda x. x^2(3) \quad \longrightarrow \quad x^2\{3/x\}
\]

C. Varela
Actor Primitives

- **send\((a, v)\)**
  - Sends value \(v\) to actor \(a\).

- **new\((b)\)**
  - Creates a new actor with behavior \(b\) (a \(\lambda\)-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

\[ b_5 = \text{rec}(\lambda y. \lambda x. \text{seq}(\text{send}(x, 5), \text{ready}(y))) \]
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 \( y \) \((b_5)\).

Sample usage:
\[ \text{send}(\text{new}(b_5), a) \]

A \textit{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**

\[ k = \alpha \parallel \mu \]

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

\( \mu \) is a multi-set of messages “en-route.”
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 \textit{continuation}) is represented as the surrounding expression with a \textit{hole} replacing the redex.

\[ \text{send}(\text{new}(b5),a) = \text{send}(\square,a) \text{\shortrightarrow new}(b5) \text{\shortleftarrow} \]

\[ e = R \text{\shortrightarrow} r \text{\shortleftarrow} \quad \text{where} \]

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

\[ r = \text{new}(b5) \]
Operational Semantics of Actors

\[
\frac{e \rightarrow^* e'}{\alpha, [R \triangleright e \triangleleft]_a \parallel \mu \quad [\text{fun}:a] \quad \alpha, [R \triangleright e' \triangleleft]_a \parallel \mu}
\]

\[
\alpha, [R \triangleright \text{new}(b) \triangleleft]_a \parallel \mu \quad [\text{new}:a,a'] \quad \alpha, [R \triangleright a' \triangleleft]_a, [\text{ready}(b)]_{a'} \parallel \mu
\]

\[a' \text{ fresh}\]

\[
\alpha, [R \triangleright \text{send}(a', v) \triangleleft]_a \parallel \mu \quad [\text{snd}:a] \quad \alpha, [R \triangleright \text{nil} \triangleleft]_a \parallel \mu \uplus \{<a' \leftarrow v>\}
\]

\[
\alpha, [R \triangleright \text{ready}(b) \triangleleft]_a \parallel \{<a \leftarrow v>\} \uplus \mu \quad [\text{rcv}:a,v] \quad \alpha, [b(v)]_a \parallel \mu
\]
Operational semantics example (1)

\[ k_0 = [\text{send}(\square, a) \triangleright \text{new}(b5) \blacktriangleright ]_a \parallel \{\} \]
\[ k_1 = [\text{send}(b, a)]_a, [\text{ready}(b5)]_b \parallel \{\} \]

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

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

\[ k_1 \xrightarrow{[\text{snd}: a]} k_2 \]
Operational semantics example (2)

\[ k_2 = [\text{nil}]_a, [\text{ready}(b5)]_b \parallel \{ < b <= a > \} \]
\[ k_3 = [\text{nil}]_a, \]
\[ [\text{rec}(\lambda y. \lambda x. \text{seq}(\text{send}(x, 5), \text{ready}(y)))(a)]_b \]
\[ \parallel \{ \} \]

\[ k_2 \xrightarrow{[\text{rcv}: b, a]} k_3 \]
\[ k_4 = [\text{nil}]_a, [\text{seq}(\text{send}(a, 5), \text{ready}(b5)))]_b \]
\[ \parallel \{ \} \]

\[ k_3 \xrightarrow{[\text{fun}: b]} k_4 \]

C. Varela
Operational semantics example (3)

\[ k_4 = [\text{nil}]_a, \]
\[ [\text{seq}(\mathbb{I}, \text{ready(b5)}) \triangleright \text{send}(a, 5) \triangleleft ]_b \]
\[ \parallel \{\} \]

\[ \begin{array}{c}
\text{[snd:b]} \\
\hline
k_4 \rightarrow k_5
\end{array} \]

\[ k_5 = [\text{nil}]_a, [\text{seq}(\text{nil}, \text{ready(b5)})]_b \]
\[ \parallel \{< a \leq 5 >\} \]
k_5 = \{\text{nil}\}_a, [\text{seq}(\text{nil}, \text{ready}(b5))]_b \\
\parallel \{< a \leq 5 >\}
\k_6 = \{\text{nil}\}_a, [\text{ready}(b5)]_b \parallel \{< a \leq 5 >\}

\begin{align*}
\k_5 & \xrightarrow{\text{fun:b}} \k_6
\end{align*}
Semantics example summary

\[ k_0 = [\text{send}(\text{new}(b5), a)]_a \ || \ {} \]
\[ k_6 = [\text{nil}]_a, [\text{ready}(b5)]_b \ || \ \{ < a \leq 5 \} \]

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

cell = rec(\lambda b. \lambda c. \lambda m. 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

\[ k_0 = [\text{ready(cell(0))}]_a \]
\[ \parallel \{<a<=s(7)>, <a<=s(2)>, <a<=g(c)>\} \]

Three receive transitions are enabled at \( k_0 \).

Multiple enabled transitions can lead to \textit{nondeterministic} behavior

The set of all computations sequences from \( k_0 \) is called the \textit{computation tree} \( \tau(k_0) \).
Nondeterministic behavior (1)

\[ k_0 = [\text{ready(cell(0))}]_a \]
\[ \quad \| \{<a<=s(7)>, <a<=s(2)> , <a<=g(c)>\} \]

\[ k_1 \rightarrow^* [\text{ready(cell(7))}]_a \]
\[ \quad \| \{<a<=s(2)> , <a<=g(c)>\} \]

\[ k_1' \rightarrow^* [\text{ready(cell(2))}]_a \]
\[ \quad \| \{<a<=s(7)> , <a<=g(c)>\} \]

\[ k_1'' \rightarrow^* [\text{ready(cell(0))}]_a \]
\[ \quad \| \{<a<=s(7)> , <a<=s(2)> , <c<=0>\} \]

Customer \(c\) will get 2 or 7.

Customer \(c\) will get 0.
Nondeterministic behavior (2)

\[ k_0 = [\text{ready(cell(0))}] \quad a \quad || \quad \{<a<=s(7)>, \quad <a<=s(2)>, \quad <a=g(c)>\} \]

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

\[
\begin{align*}
\text{rcv:}\ a,g(c) & \quad \rightarrow \quad \text{rcv:}\ a,s(7) & \quad \rightarrow \quad \text{rcv:}\ a,s(2) \\
k_0 & \quad \rightarrow \quad k_1 & \quad \rightarrow \quad k_2 & \quad \rightarrow \quad k_3
\end{align*}
\]

\[ k_f = [\text{ready(cell(2))}] \quad a \quad || \quad \{<c<=0>\} \]

**Final cell state is 2.**
Nondeterministic behavior (3)

\[ k_0 = [\text{ready(cell(0))}]_a \]
\[ \parallel \{<a<=s(7)>, <a<=s(2)>, <a<=g(c)>\} \]

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

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

\[ k_f = [\text{ready(cell(7))}]_a \parallel \{<c<=2>\} \]

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) ->
    receive
        {set, NewContent} -> cell(NewContent);
        {get, Customer} -> Customer ! Content,
                          cell(Content)
    end.

Encapsulated state Content.
Message handlers

State change.
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.
-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 Model
  – A reasoning framework to model concurrent computations
  – Programming abstractions for distributed open systems


• 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

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.
module cell;

behavior Cell {
    Object content;

    Cell(Object initialContent) {
        content = initialContent;
    }

    Object get() { return content; }

    void set(Object newContent) {
        content = newContent;
    }
}
module cell;

behavior Cell {
    
    Object content;

    Cell(Object initialContent) {
        content = initialContent;
    }

    Object get() { return content; }

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

Encapsulated state content.

Actor constructor.

Message handlers.

State change.
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

```java
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 );
    }
}
```
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.
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.
Consider:

\[
\text{treeprod} = \text{rec}(\lambda f. \lambda \text{tree}.
\begin{align*}
&\quad \text{if}(\text{isnat}(\text{tree}), \\
&\quad \quad \text{tree}, \\
&\quad \quad f(\text{left}(\text{tree})) \times f(\text{right}(\text{tree}))))
\end{align*}
\]

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

You can do the “left” and “right” computations concurrently.
Tree Product Behavior in AMST

\[
B_{\text{treeprod}} = \\
\text{rec}(\lambda b. \lambda m. \\
\quad \text{seq}(\text{if}(\text{isnat}(\text{tree}(m))), \\
\quad \quad \text{send}(\text{cust}(m), \text{tree}(m)), \\
\quad \quad \text{let } \text{newcust} = \text{new}(B_{\text{joincont}}(\text{cust}(m))), \\
\quad \quad \quad \text{lp} = \text{new}(B_{\text{treeprod}}), \\
\quad \quad \quad \text{rp} = \text{new}(B_{\text{treeprod}}) \text{ in} \\
\quad \quad \text{seq}(\text{send}(\text{lp}, \\
\quad \quad \quad \text{pr}(\text{left}(\text{tree}(m)), \text{newcust}))), \\
\quad \quad \text{send}(\text{rp}, \\
\quad \quad \quad \text{pr}(\text{right}(\text{tree}(m)), \text{newcust}))), \\
\quad \quad \text{ready}(b))))
\]
Join Continuation in AMST

\[ B_{\text{joincont}} = \lambda \text{cust.} \lambda \text{firstnum.} \text{ready}(\lambda \text{num.}
\text{seq}(\text{send}(\text{cust}, \text{firstnum} \times \text{num}),
\text{ready}(\text{sink}))) \]
Sample Execution

(a) \( f(\text{tree}, \text{cust}) \)

(b) \( f(\text{left(tree)}, \text{JC}) \)

\( f(\text{right(tree)}, \text{JC}) \)
Sample Execution

\( f(\text{left(tree)}, \text{JC}) \)

(c)

(d)
Sample Execution
Tree Product Behavior in Erlang

-module(treeprod).
-export([[treeprod/0, join/1]]).

treeprod() ->
    receive
        {{Left, Right}, Customer} ->
            NewCust = spawn(treeprod, join, [Customer]),
            LP = spawn(treeprod, treeprod, []),
            RP = spawn(treeprod, treeprod, []),
            LP!{Left, NewCust},
            RP!{Right, NewCust};
        {Number, Customer} ->
            Customer ! Number
    end,
    treeprod().

join(Customer) -> receive V1 -> receive V2 -> Customer ! V1*V2 end end.
Tree Product Sample Execution

2> TP = spawn(treeprod,treeprod,[]).
<0.40.0>
3> TP ! {{{{5,6},2},{3,4}},self()}. 
{{{{{5,6},2},{3,4}}},<0.33.0>}
4> flush().
Shell got 720
ok
5>
module treeprod;
import tree.Tree;

behavior TreeProduct {
  int multiply(Object[] results){
    return (Integer) results[0] * (Integer) results[1];
  }
  int compute(Tree t){
    if (t.isLeaf()) return t.value();
    else {
      TreeProduct lp = new TreeProduct();
      TreeProduct rp = new TreeProduct();
      join {
        lp <- compute(t.left());
        rp <- compute(t.right());
      } @ multiply(token) @ currentContinuation;
    }
  }
}

This code uses token-passing continuations (@,token), a join block (join), and a first-class continuation (currentContinuation).
module treeprod;
import tree.Tree;

behavior TreeProductTester {
    void act(String[] args) {
        Tree t = new Tree(new Tree(new Tree(5, 6), new Tree(2)),
                          new Tree(3, 4));
        TreeProduct tp = new TreeProduct();
        tp <- compute(t) @ standardOutput <- println(token);
    }
}

Use as follows:
% javac tree/Tree.java
% salsac treeprod/*
% salsa treeprod/TreeProductTester
720

C. Varela
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. PDCS Exercise 4.6.1 (page 77).

43. Modify the `treeprod` behavior in Erlang to reuse the tree product actor to compute the product of the left subtree. (See PDCS page 63 for the corresponding \(tprod_2\) behavior in AMST.)

44. PDCS Exercise 9.6.1 (page 203). Modify your code as in Exercise 43.

45. Create a concurrent `fibonacci` behavior in Erlang using join continuations, and in SALSA using a join block.