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

Support for the actor model in SALSA and Erlang

Carlos Varela

Rennselaer Polytechnic Institute

October 11, 2019

* 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

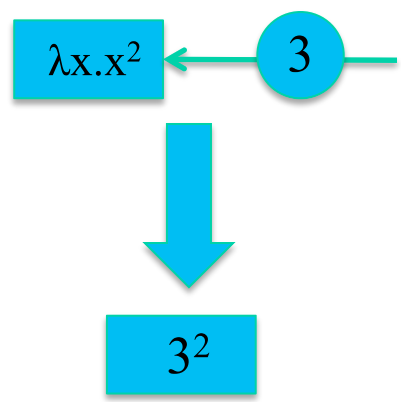
Syntax:

`e`	`::=`	`v`	*variable*
	|	`λv.e`	*function*
	|	`e(e)`	*application*

Example of beta-reduction:

$\lambda x.x^2(3)$

$\longrightarrow \quad x^2\{3/x\}$



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 = rec(λy.λx.seq(send(x,5),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:

```
send(new(b5), a)
```

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

```
sink = rec(λb.λm.ready(b))
```

Operational Semantics for AMST Actor Language

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

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

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

Actor Configurations

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

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

$\boldsymbol{\mu}$ is a multi-set of messages "en-route."

Reduction contexts and redexes

Consider the expression:

`e = send(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.

`send(new(b5),a) = send(□,a)▶new(b5)◀`

`e = R▶r◀` where

`R = send(□,a)`

`r = new(b5)`

Operational Semantics of Actors

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

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

a' fresh

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

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

Operational semantics example (1)

k_0 = [send(☐,a)►new(b5)◄]$_a$ || {}

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

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

k_2 = [nil]$_a$, [ready(b5)]$_b$ || {< *b* <= *a* >}

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

Operational semantics example (2)

k_2 = [nil]$_a$, [ready(b5)]$_b$ || {< *b* <= *a* >}

k_3 = [nil]$_a$,

[rec(λy.λx.seq(send(x,5),ready(y)))(a)]$_b$

|| {}

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

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

|| {}

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

Operational semantics example (3)

k_4 = [nil]$_a$,
[seq(□,ready(b5))▶ send(a,5) ◀]$_b$
|| {}

$$k_4 \xrightarrow{[\mathbf{snd}: a, 5]} k_5$$

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

Operational semantics example (4)

$$k_5 = [\texttt{nil}]_a, [\texttt{seq(nil,ready(b5))}]_b$$
$$\parallel \; \{\langle\, a \Leftarrow 5 \,\rangle\}$$

$$k_6 = [\texttt{nil}]_a, [\texttt{ready(b5)}]_b \; \parallel \; \{\langle\, a \Leftarrow 5 \,\rangle\}$$

$$\mathbf{k_5} \xrightarrow{[\mathbf{fun}:b]} \mathbf{k_6}$$

Semantics example summary

k_0 = [send(new(b5),a)]$_a$ || {}

k_6 = [nil]$_a$, [ready(b5)]$_b$ || {< a <= 5 >}

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

$$\mathbf{k_4} \xrightarrow{[\mathbf{snd}:a,5]} \mathbf{k_5} \xrightarrow{[\mathbf{fun}:b]} \mathbf{k_6}$$

This sequence of (labeled) transitions from $\mathbf{k_0}$ to $\mathbf{k_6}$ 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

```
k0 = [ready(cell(0))]a
       || {<a<=s(7)>, <a<=s(2)>, <a<=g(c)>}
```

Three receive transitions are enabled at k_0.

> Multiple enabled transitions can lead to *nondeterministic* behavior

$$\mathbf{k}_0 \xrightarrow{[\mathbf{rcv}:a,s(7)]} \mathbf{k}_1$$

$$\mathbf{k}_0 \xrightarrow{[\mathbf{rcv}:a,s(2)]} \mathbf{k}_1'$$

$$\mathbf{k}_0 \xrightarrow{[\mathbf{rcv}:a,g(c)]} \mathbf{k}_1''$$

> The set of all computations sequences from $\mathbf{k}_0$ is called the *computation tree* $\tau(\mathbf{k}_0)$.

Nondeterministic behavior (1)

$k_0 = [ready(cell(0))]_a$

$\parallel \{\langle a<=s(7)\rangle, \langle a<=s(2)\rangle, \langle a<=g(c)\rangle\}$

$k_1 \rightarrow^* [ready(cell(7))]_a$

$\parallel \{\langle a<=s(2)\rangle, \langle a<=g(c)\rangle\}$

Customer c will get 2 or 7.

$k_1' \rightarrow^* [ready(cell(2))]_a$

$\parallel \{\langle a<=s(7)\rangle, \langle a<=g(c)\rangle\}$

Customer c will get 0.

$k_1'' \rightarrow^* [ready(cell(0))]_a$

$\parallel \{\langle a<=s(7)\rangle, \langle a<=s(2)\rangle, \langle c<=0\rangle\}$

Nondeterministic behavior (2)

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

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

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

$k_f = [\mathtt{ready(cell(2))}]_a \parallel \{<c<=0>\}$

Final cell state is **2**.

Nondeterministic behavior (3)

k_0 = [ready(cell(0))]$_a$

‖ {<a<=s(7)>, <a<=s(2)>, <a<=g(c)>}

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

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

k_f = [ready(cell(7))]$_a$ ‖ {<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.**

State change.

Message handlers

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)`

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

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

Actors/SALSA

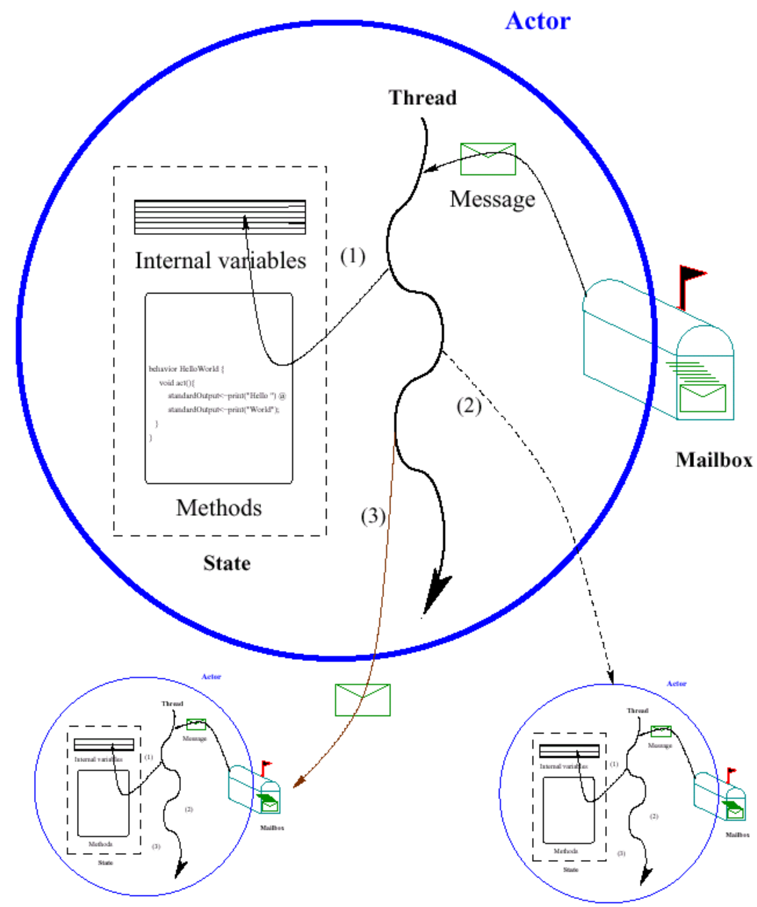
- Actor Model
  - A reasoning framework to model concurrent computations
  - Programming abstractions for distributed open systems

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;

   Cell(Object initialContent) {
         content = initialContent;
    }

   Object get() { return content; }

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

Encapsulated state content.

Actor constructor.

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.

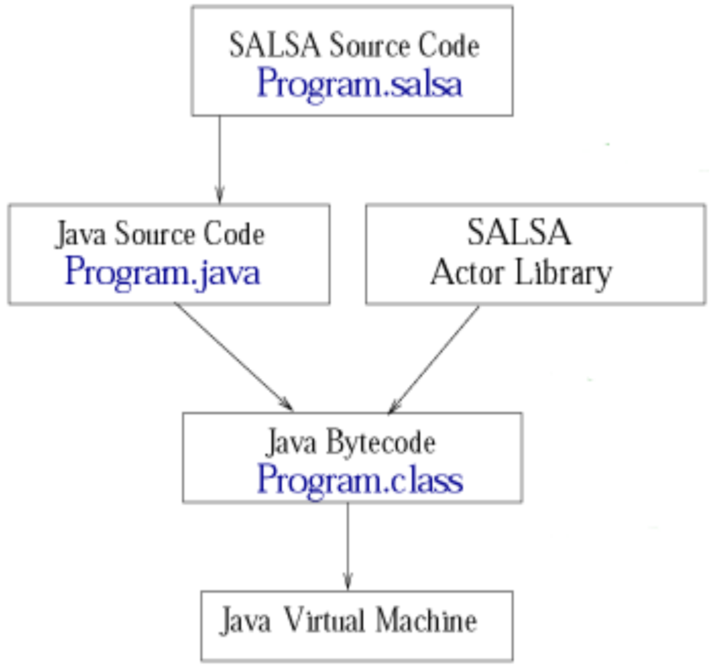
Cell Tester Example

```
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.

Join Continuations

Consider:

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

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

You can do the “left” and “right” computations concurrently.

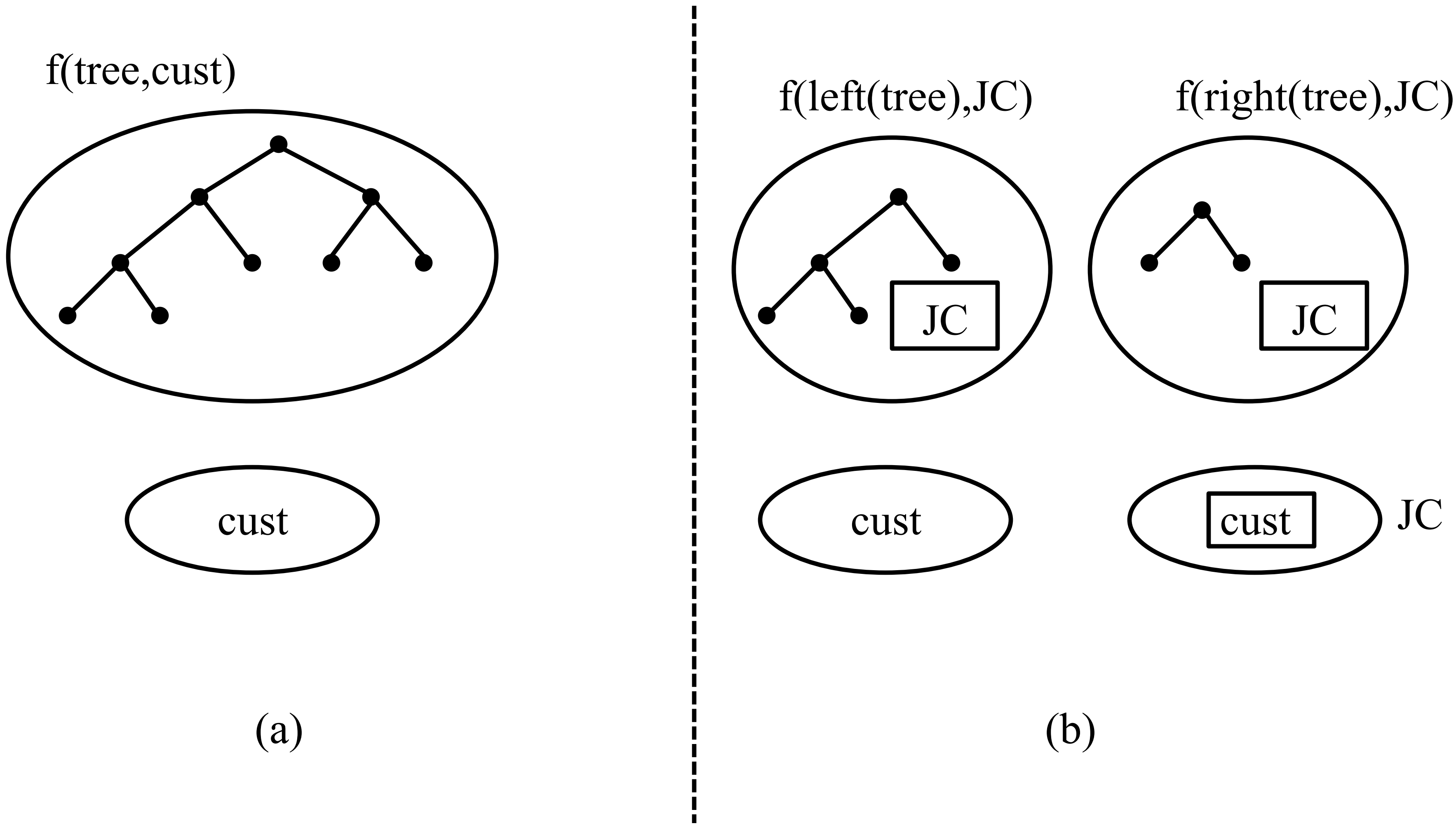
Tree Product Behavior in AMST

```
Btreeprod =
  rec(λb.λm.
      seq(if(isnat(tree(m)),
             send(cust(m),tree(m)),
             let newcust=new(Bjoincont(cust(m))),
                  lp = new(Btreeprod),
                  rp = new(Btreeprod) in
             seq(send(lp,
                  pr(left(tree(m)),newcust)),
                 send(rp,
                  pr(right(tree(m)),newcust)))),
          ready(b)))
```

Join Continuation in AMST

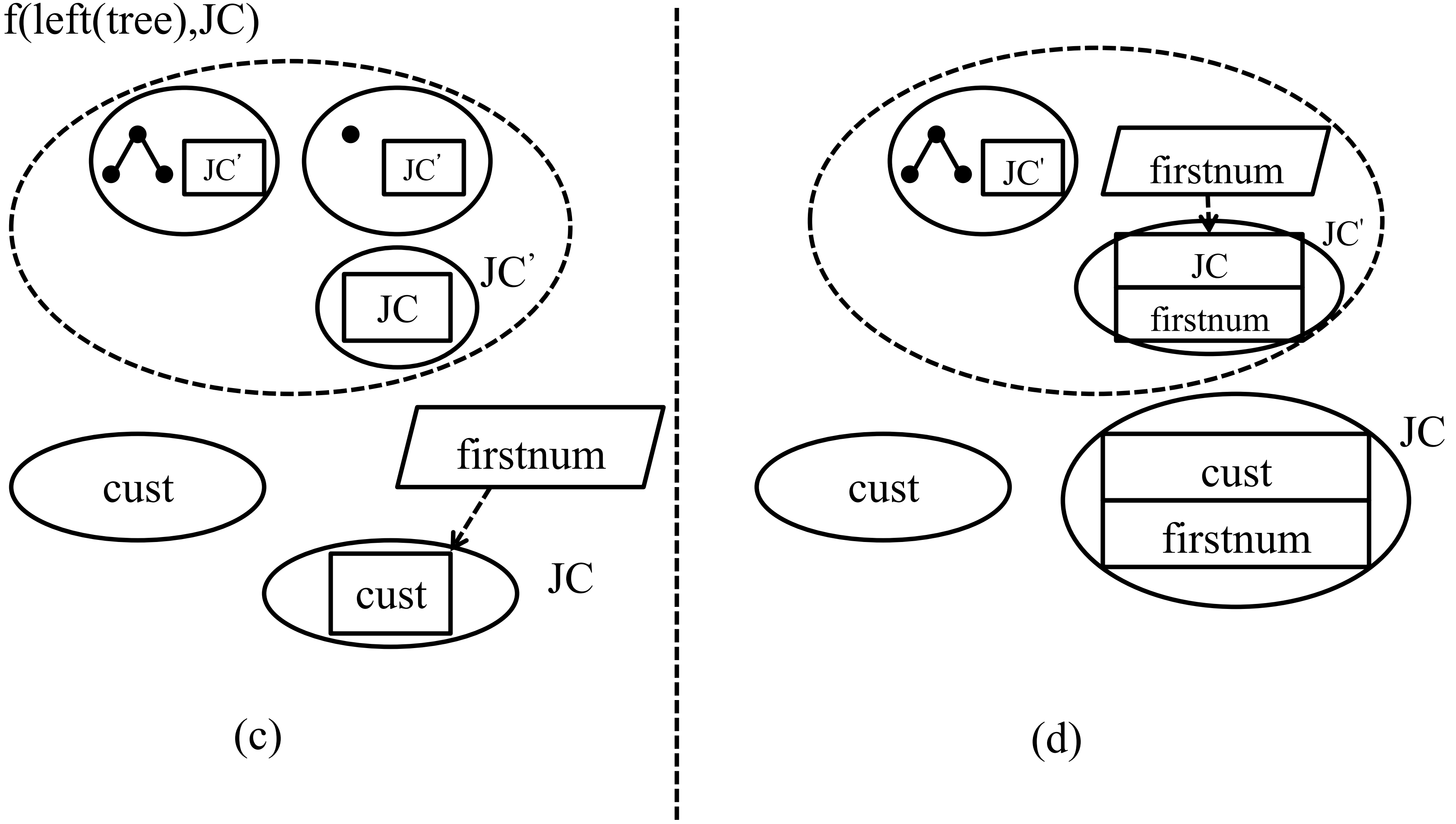
```
Bjoincont =
  λcust.λfirstnum.ready(λnum.
            seq(send(cust,firstnum*num),
                   ready(sink)))
```

Sample Execution

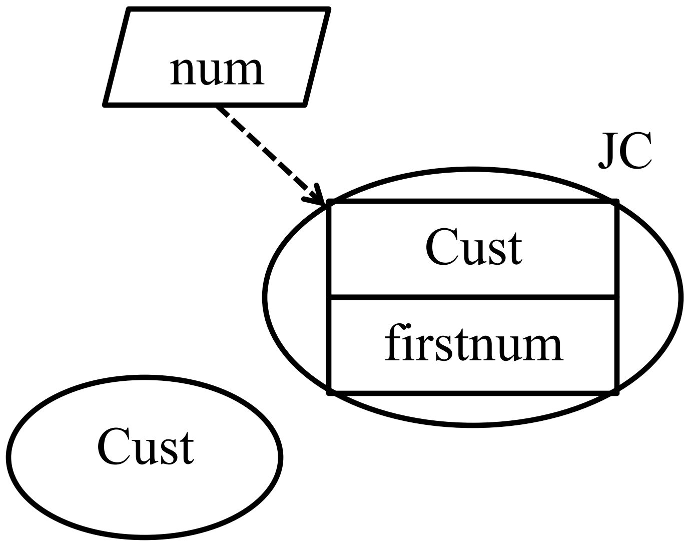


Sample Execution

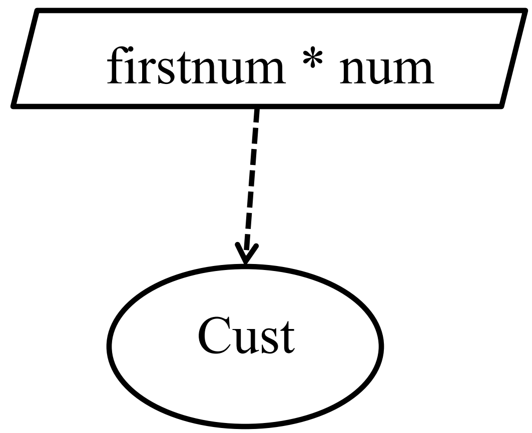
f(left(tree),JC)



Sample Execution



(e)



(f)

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>
```

Tree Product Behavior in SALSA

This code uses token-passing continuations (`@`,`token`), a join block (`join`), and a first-class continuation (`currentContinuation`).

```
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;
      }
    }
}
```

Tree Product Tester

```
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
```

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 $\texttt{tprod}_2$ 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.