

Actors (PDCS 4)

AMST actor language syntax, semantics, join
continuations

Carlos Varela

Rensselaer Polytechnic Institute

October 2, 2026

Advantages of concurrent programs

- **Reactive programming**
 - User can interact with applications while tasks are running, *e.g.*, stopping the transfer of a large file in a web browser.
- **Availability of services**
 - Long-running tasks need not delay short-running ones, *e.g.*, a web server can serve an entry page while at the same time processing a complex query.
- **Parallelism**
 - Complex programs can make better use of hardware resources in multi-core processor architectures, SMPs, LANs, WANs, grids, and clouds, *e.g.*, scientific/engineering applications, simulations, games, etc.
- **Controllability**
 - Tasks requiring certain preconditions can suspend and wait until the preconditions hold, then resume execution transparently.

Disadvantages of concurrent programs

- **Safety**
 - « *Nothing bad ever happens* »
 - Concurrent tasks should not corrupt consistent state of program.
- **Liveness**
 - « *Anything ever happens at all* »
 - Tasks should not suspend and indefinitely wait for each other (deadlock).
- **Non-determinism**
 - Mastering exponential number of interleavings due to different schedules.
- **Resource consumption**
 - Concurrency can be expensive. Overhead of scheduling, context-switching, and synchronization.
 - Concurrent programs can run *slower* than their sequential counterparts even with multiple CPUs!

Overview of concurrent programming

- There are four main approaches:
 - Sequential programming (no concurrency)
 - Declarative concurrency (streams in a functional language)
 - Message passing with active objects (Erlang, SALSA)
 - Atomic actions on shared state (Java, C++)
- The atomic action approach is the *most difficult*, yet it is the one you will probably be most exposed to!
- But, if you have the choice, which approach to use?
 - Use the simplest approach that does the job: sequential if that is ok, else declarative concurrency if there is no observable nondeterminism, otherwise use actors and message passing.

Actors/SALSA

- Actor Model

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

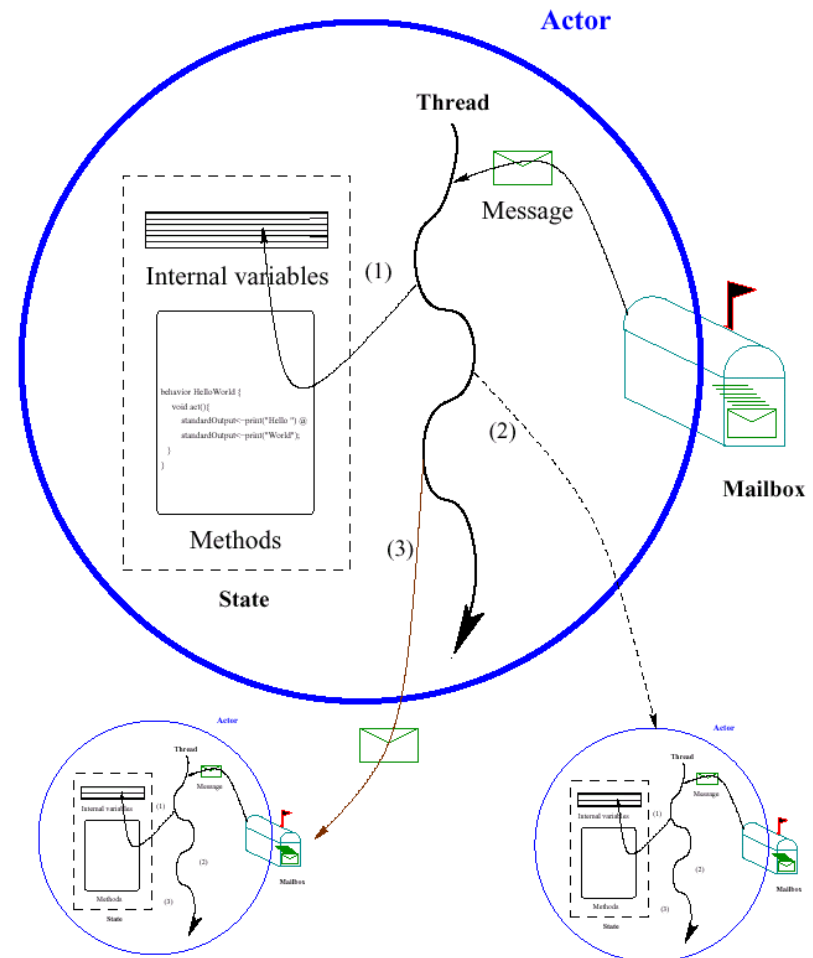
G. Agha, *Actors: A Model of Concurrent Computation in Distributed Systems*. MIT Press, 1986.

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.



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.

Open Distributed Systems

- Addition of new components
- Replacement of existing components
- Changes in interconnections

Synchronous vs. Asynchronous Communication

- The π -calculus (and other process algebras such as CCS, CSP) uses synchronous communication.
- The actor model assumes asynchronous communication is *the most* primitive interaction mechanism.

Communication Medium

- In the π -calculus, channels are explicitly modeled. Multiple processes can share a channel, potentially causing interference.
- In the actor model, the communication medium is not explicit. Actors (active objects) are first-class, history-sensitive (stateful) entities with an explicit identity used for communication.

Fairness

- The actor model theory assumes fair computations:
 1. Message delivery is guaranteed.
 2. Infinitely-often enabled computations must eventually happen.

Fairness is very useful for reasoning about equivalences of actor programs but can be hard/expensive to guarantee; in particular when distribution, mobility, and failures are considered.

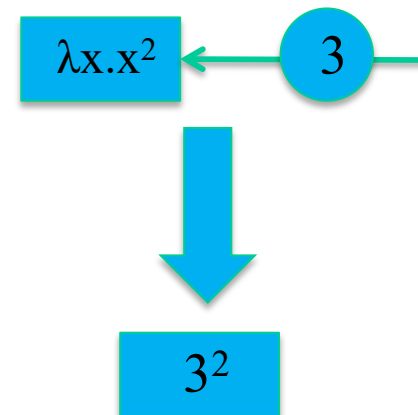
λ -Calculus as a Model for Sequential Computation

Syntax:

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

Example of beta-reduction:

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



λ -Calculus extended with pairs

- $\text{pr}(x, y)$ *returns a pair containing x & y*
- $\text{ispr}(x)$ *returns $\mathbf{t}$ if x is a pair; $\mathbf{f}$ otherwise*
- $1^{\text{st}}(\text{pr}(x, y)) = x$ *returns the first value of a pair*
- $2^{\text{nd}}(\text{pr}(x, y)) = y$ *returns the 2nd value of a pair*

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:

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

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)) )
```

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

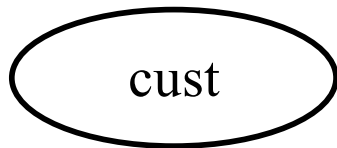
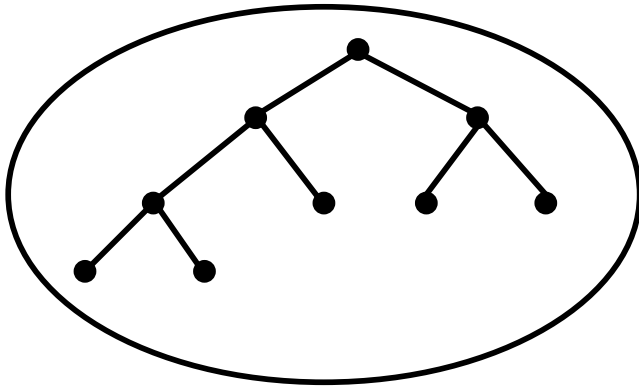
```
 $B_{treeprod} =$   
  rec ( $\lambda b. \lambda m.$   
    seq (if (isnat (tree(m)) ,  
            send (cust(m) , tree(m)) ,  
            let newcust=new( $B_{joincont}$ (cust(m))) ,  
              lp = new(b) ,  
              rp = new(b) in  
            seq(send(lp ,  
                  pr(left(tree(m))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

Tree Product (continued)

```
 $B_{joincont} =$   
   $\lambda cust. \lambda firstnum. ready(\lambda num.$   
     $seq(send(cust, firstnum * num),$   
       $ready(sink)))$ 
```

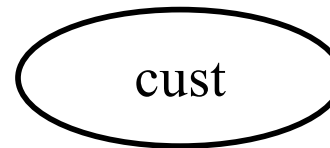
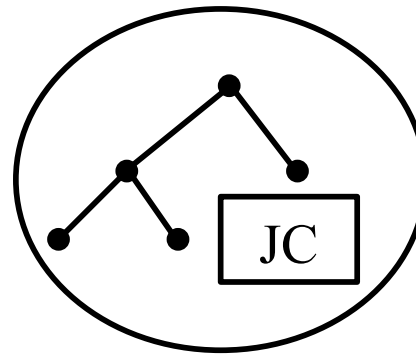
Sample Execution

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



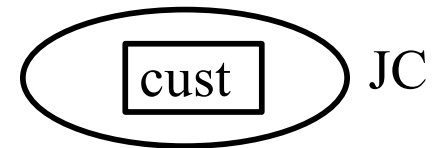
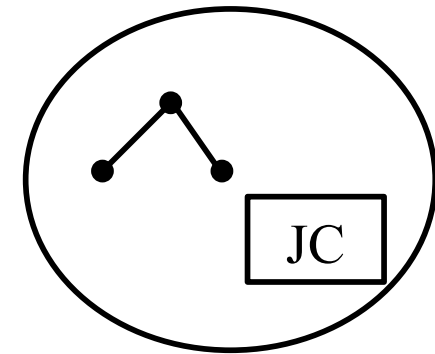
(a)

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



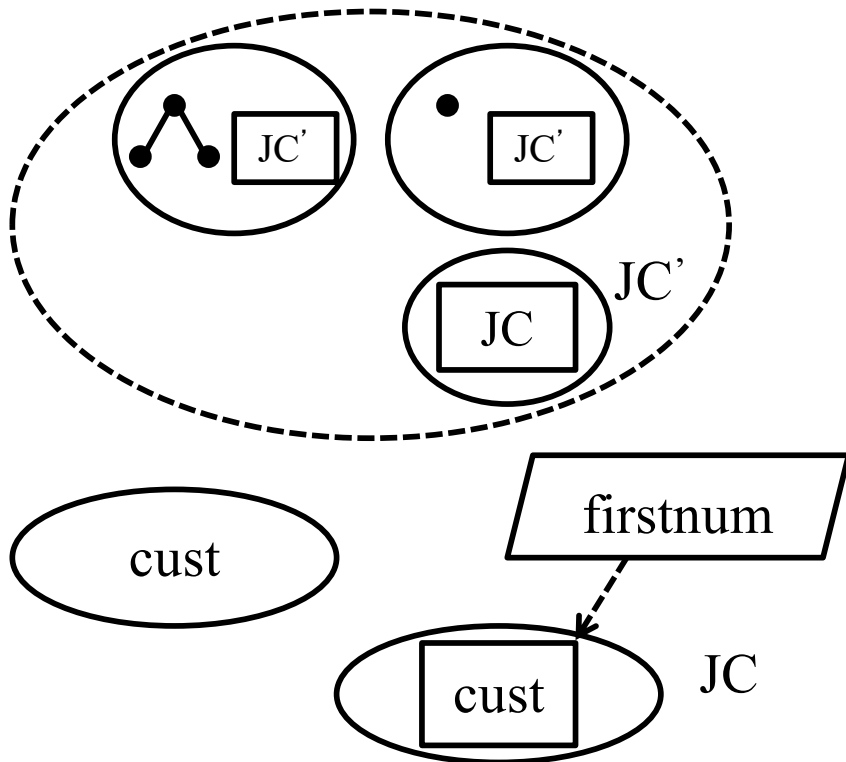
(b)

$f(\text{right}(\text{tree}), \text{JC})$

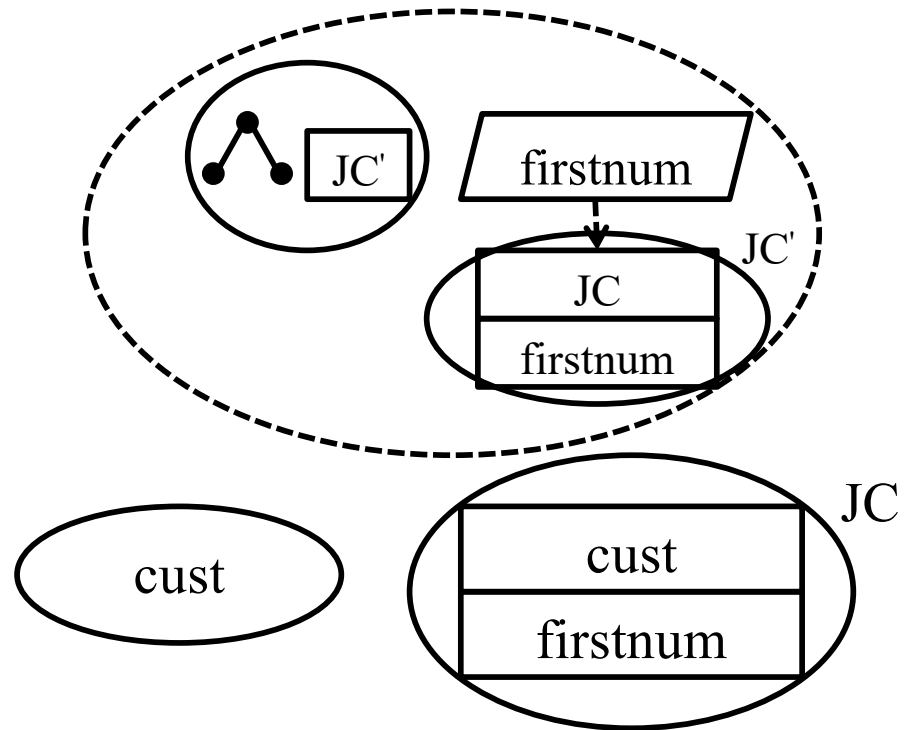


Sample Execution

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

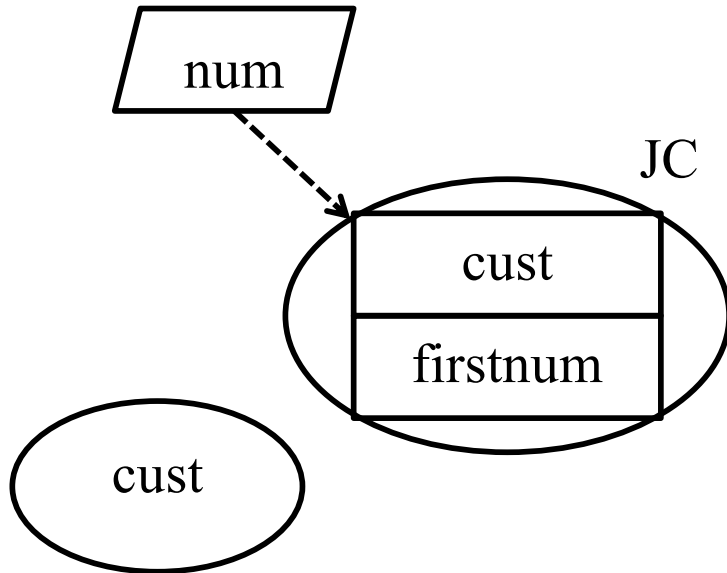


(c)

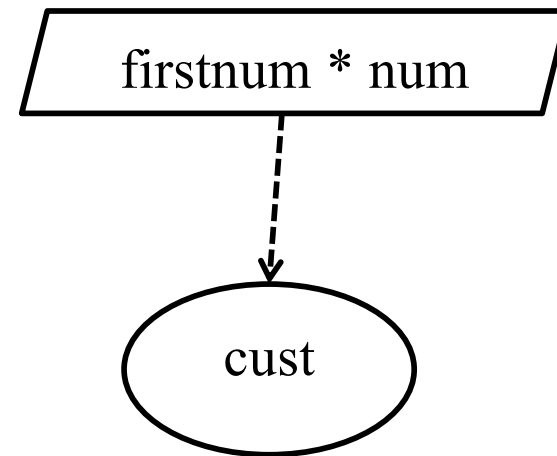


(d)

Sample Execution



(e)



(f)

Operational Semantics of AMST Actor Language

- Operational semantics of actor language 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

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

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

Labeled Transition Relation

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

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

a' fresh

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

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

Exercises

36. PDCS Exercise 4.6.1 (page 77).

37. Write get?
 cust
 set?
 contents
 mkset
 mkget

to complete the reference cell example in the AMST actor language.

38. Modify the `cell` behavior to notify a customer when the cell value has been updated.

39. PDCS Exercise 4.6.6 (page 77).

40. PDCS Exercise 4.6.7 (page 78).