Actors (PDCS 4)
AMST actor language syntax, semantics, join continuations

Carlos Varela
Rennselaer Polytechnic Institute

October 6, 2023

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

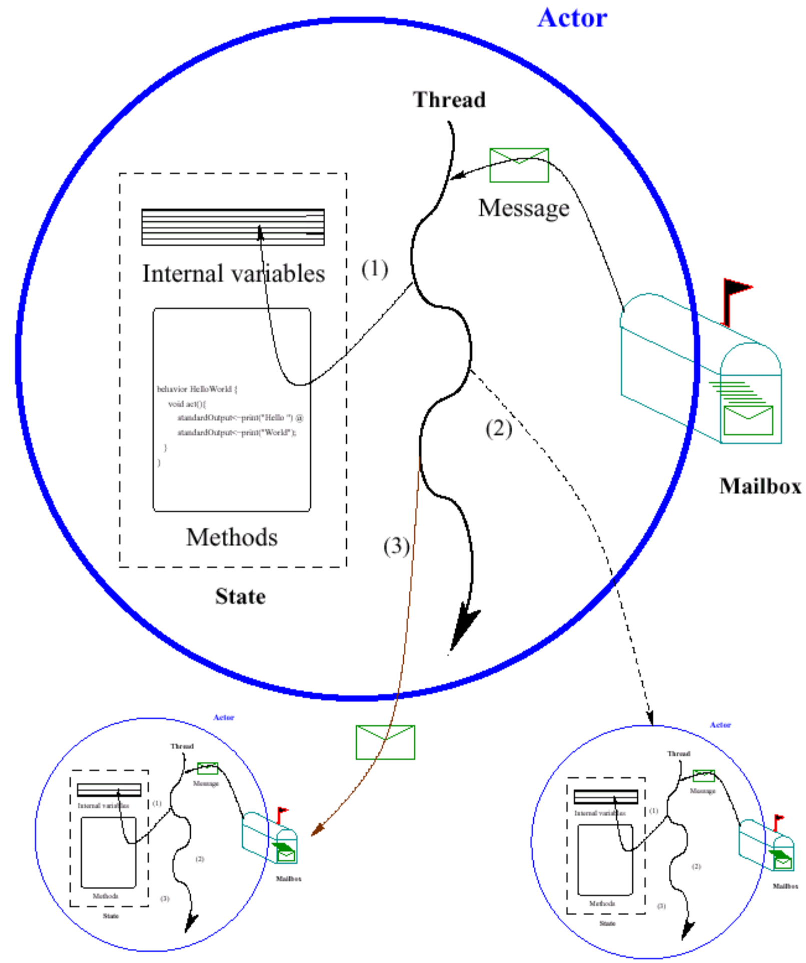
- Actor Model
  - A reasoning framework to model concurrent computations
  - Programming abstractions for distributed open systems

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 + `if`s and `pair`s) 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

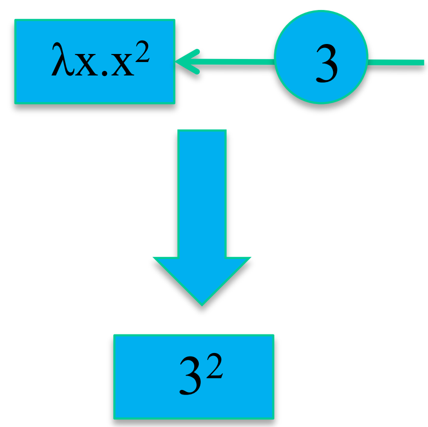
Syntax:

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

Example of beta-reduction:

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

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



λ-Calculus extended with pairs

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

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

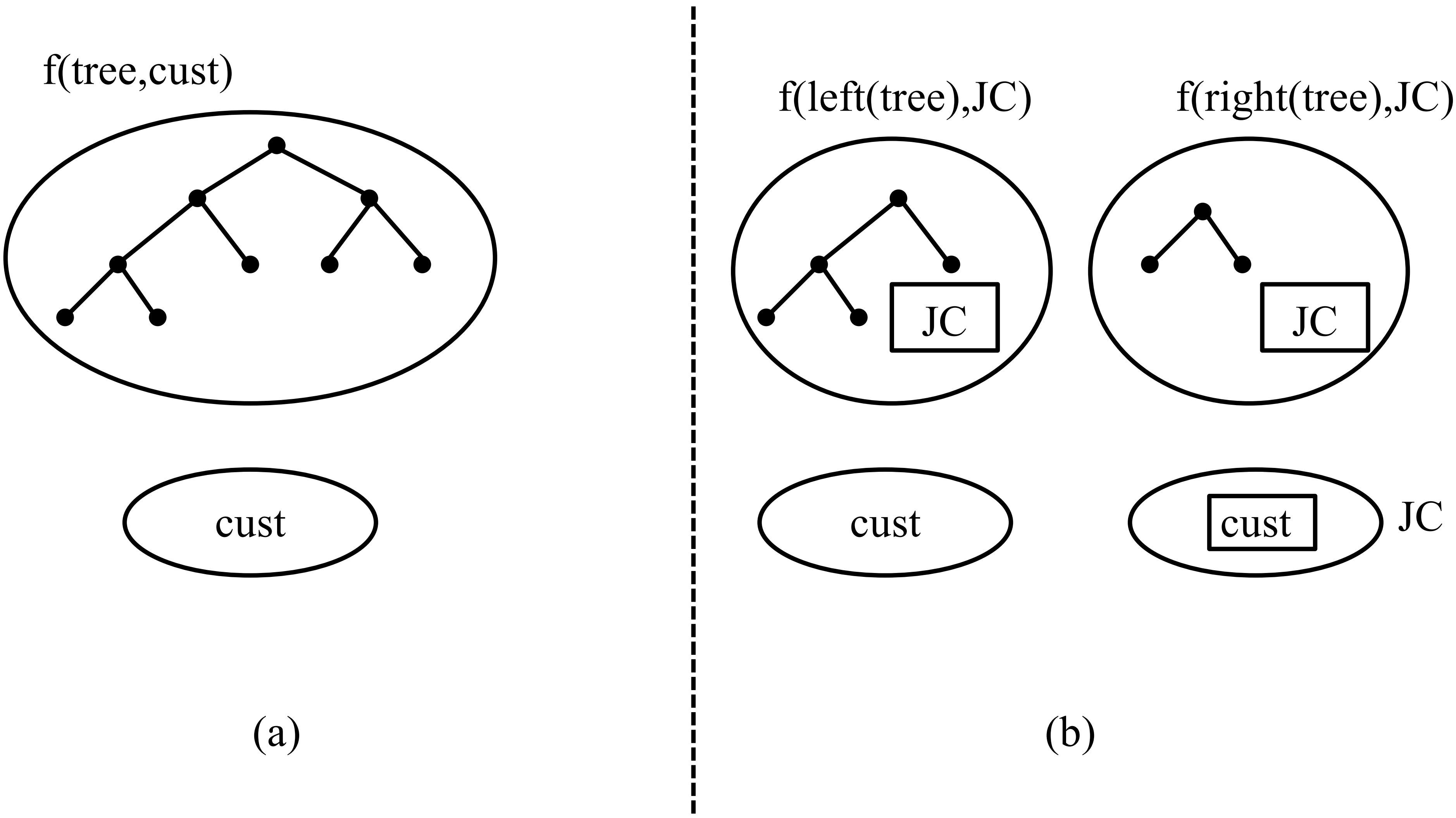
$B_{treeprod}$ =

```
rec(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)))
```

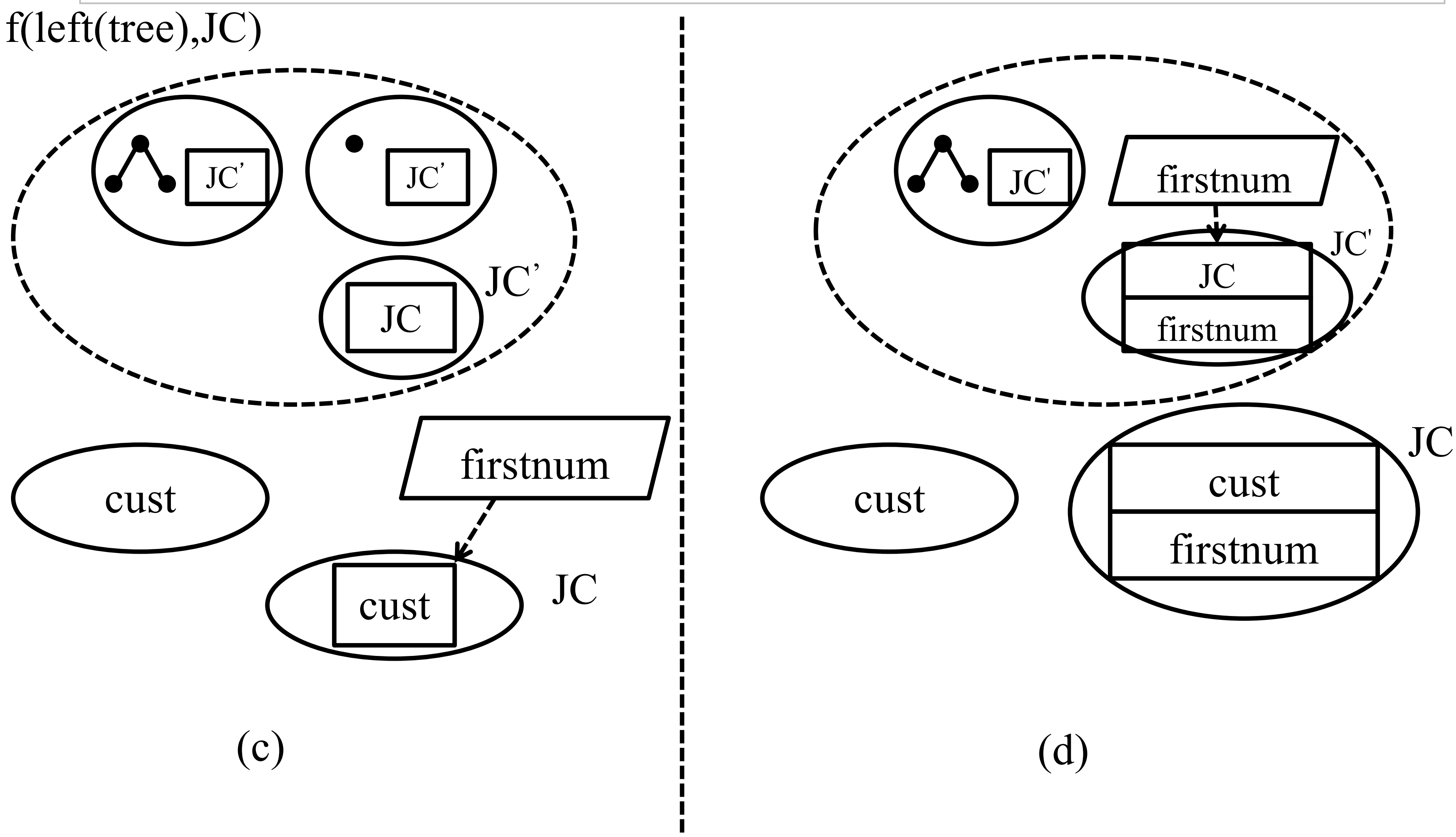
Tree Product (continued)

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

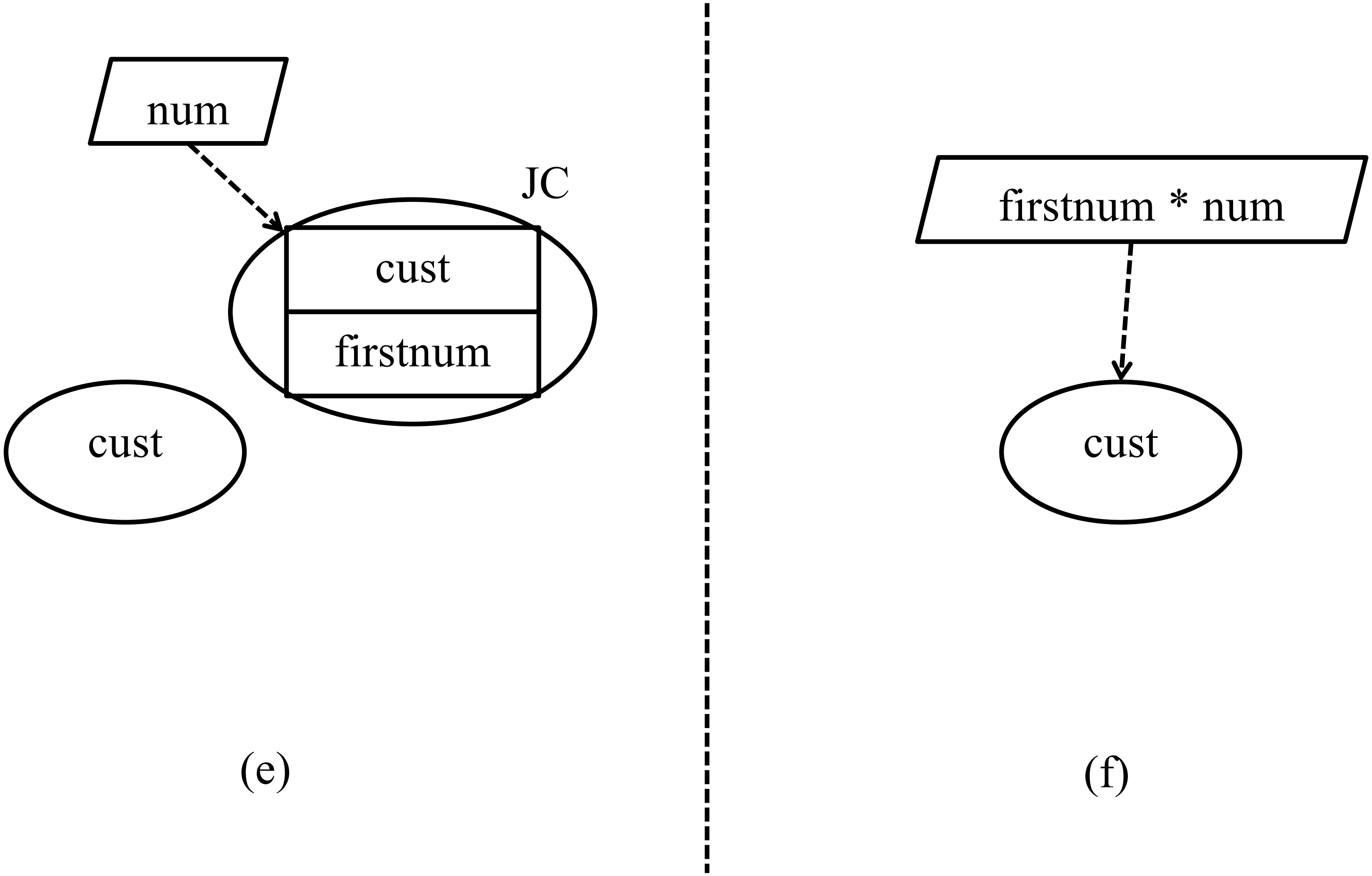
Sample Execution



Sample Execution



Sample Execution



Operational Semantics of AMST Actor Language

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

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

- 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."

Syntactic restrictions on configurations

Given $A = Dom(\alpha)$:

- If a in A, then $fv(\alpha(a))$ is a subset of A.
- If $<a <= v>$ in μ, then $\{a\} \cup fv(v)$ is a subset of A.

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)`

Labeled Transition Relation

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

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

a' *fresh*

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

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

Exercises

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).