

Concurrency control abstractions (PDCS 9, CPE 5*)

Carlos Varela
Rensselaer Polytechnic Institute

October 14, 2016

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

C. Varela

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

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

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

AMST Semantics Example

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

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

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

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

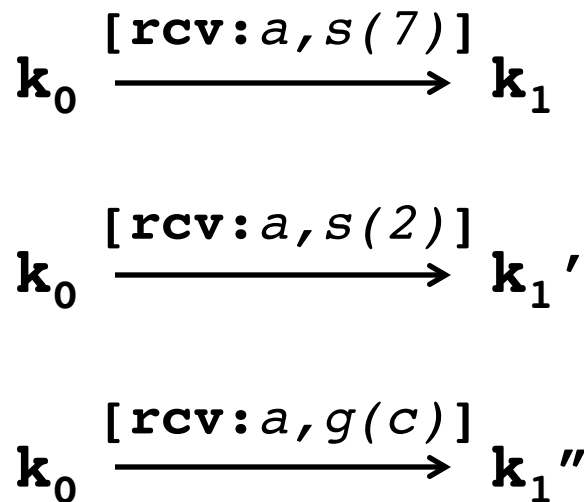
This sequence of (labeled) transitions from k_0 to k_6 is called a *computation sequence*.

Nondeterministic Behavior

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

Three receive transitions are enabled at k_0 .

Multiple enabled transitions can lead to *nondeterministic* behavior



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

Actors/SALSA

- Actor Model

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

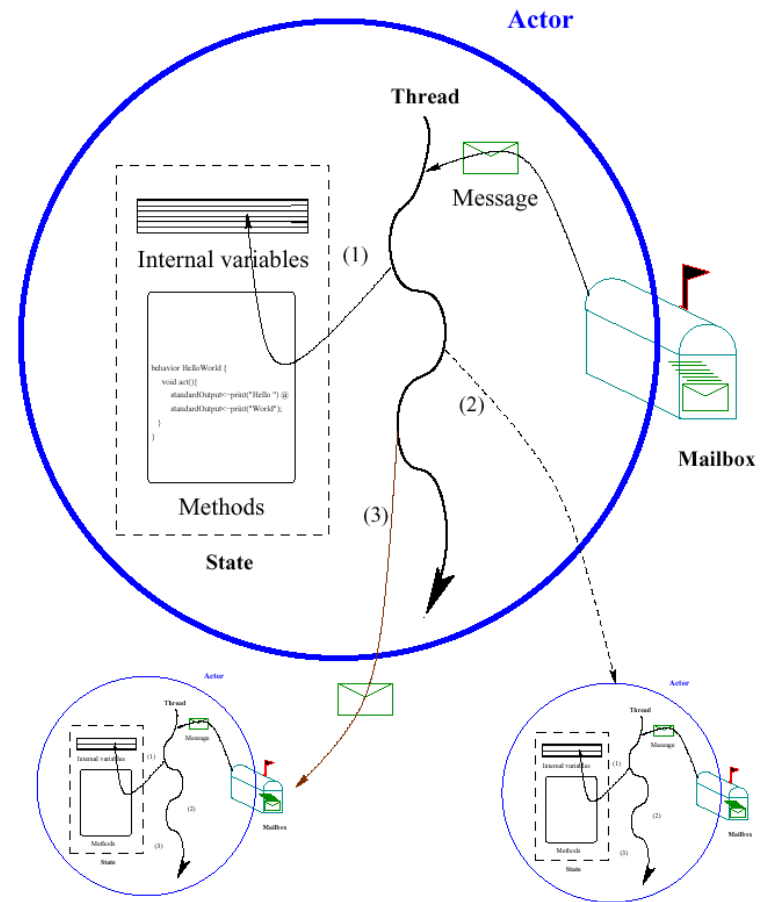
G. Agha, *Actors: A Model of Concurrent Computation in Distributed Systems*. MIT Press, 1986.

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.



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.

Cell Tester Example

```
module cell;

behavior CellTester {

    void act( String[] args ) {

        Cell c = new Cell(0);
        c <- set(2);
        c <- set(7);
        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.

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.

Cell Tester in Erlang

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

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

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

Actor creation (**spawn**)

Message passing (**!**)

receive waits until a message is available.

Tree Product Behavior in AMST

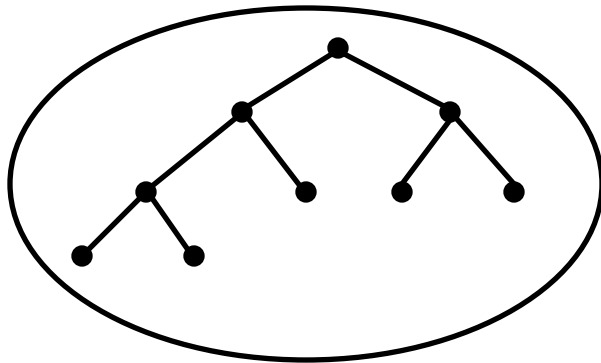
```
Btreeprod =  
  rec (λb.λm.  
    seq (if (isnat (tree (m))) ,  
          send (cust (m) , tree (m)) ,  
          let newcust=new (Bjoincont (cust (m))) ,  
              lp = new (Btreeprod) ,  
              rp = new (Btreeprod) in  
          seq (send (lp ,  
                    pr (left (tree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

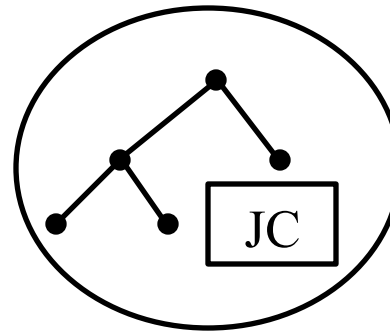
$f(\text{tree}, \text{cust})$



cust

(a)

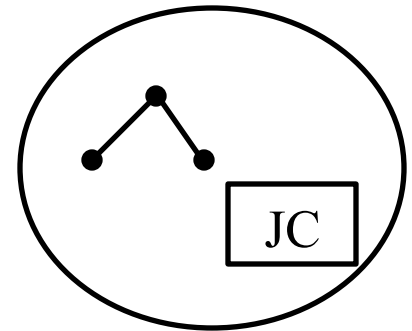
$f(\text{left}(\text{tree}), \text{JC})$



cust

(b)

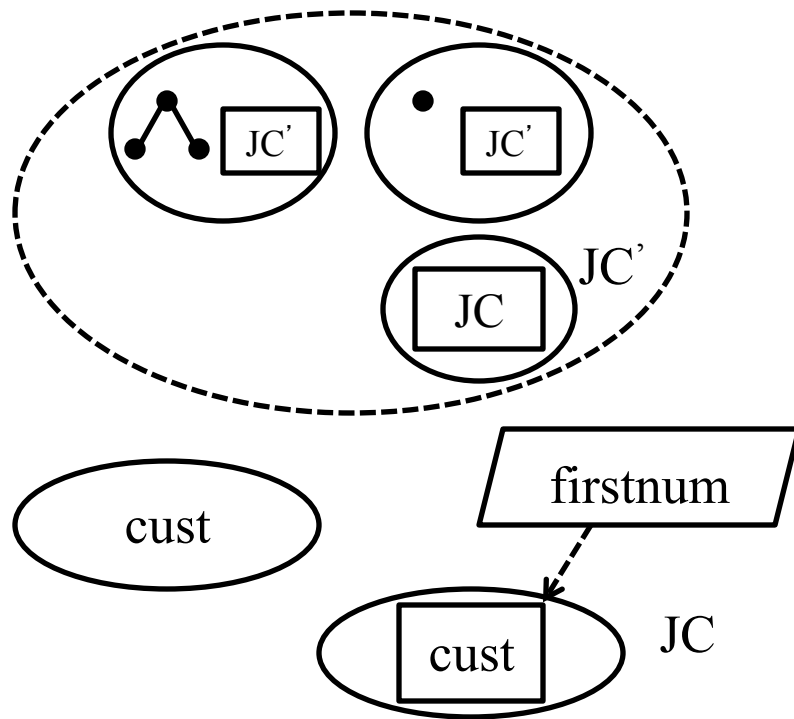
$f(\text{right}(\text{tree}), \text{JC})$



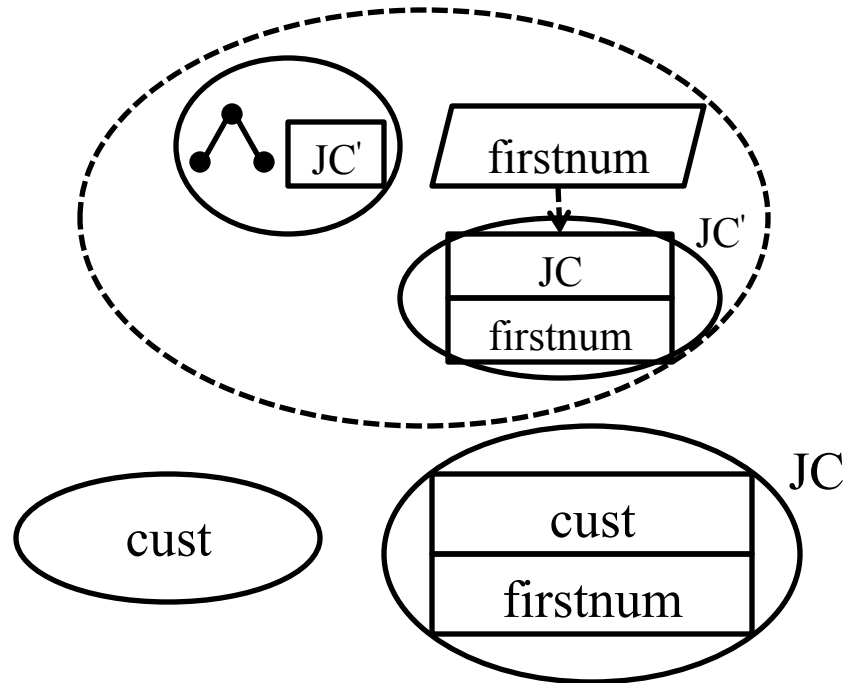
JC

Sample Execution

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

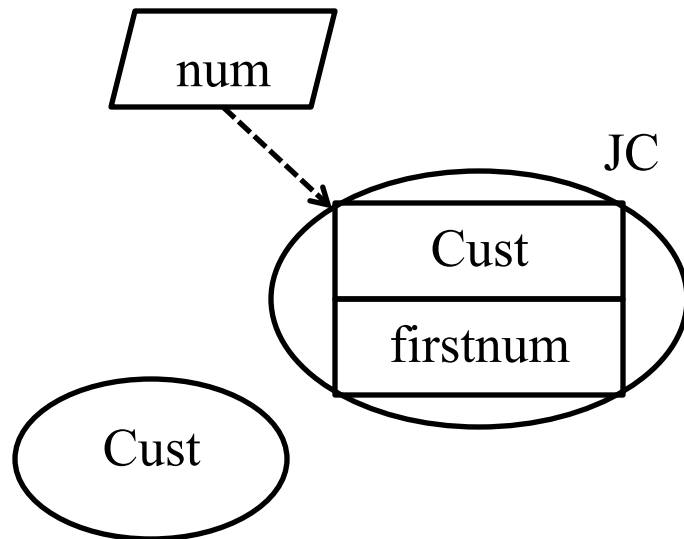


(c)

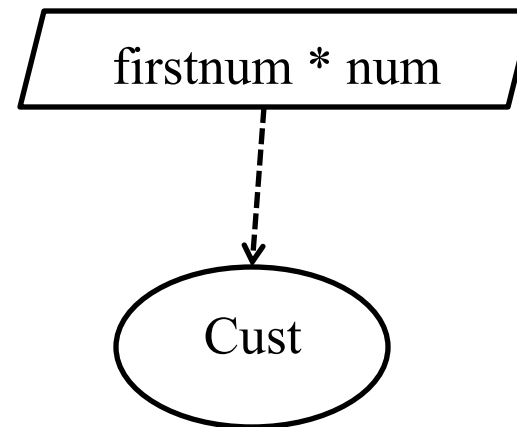


(d)

Sample Execution



(e)



(f)

Tree Product Behavior in SALSA

```
module treeprod;

behavior TreeProduct {

    void compute(Tree t, UniversalActor c){
        if (t.isLeaf()) c <- result(t.value());
        else {
            JoinCont newCust = new JoinCont(c);
            TreeProduct lp = new TreeProduct();
            TreeProduct rp = new TreeProduct();
            lp <- compute(t.left(), newCust);
            rp <- compute(t.right(), newCust);
        }
    }
}
```

Join Continuation in SALSA

```
module treeprod;
behavior JoinCont {

    UniversalActor cust;
    int first;
    boolean receivedFirst;

    JoinCont(UniversalActor cust){
        this.cust = cust;
        this.receivedFirst = false;
    }

    void result(int v) {
        if (!receivedFirst){
            first = v; receivedFirst = true;
        }
        else // receiving second value
            cust <- result(first*v);
    }
}
```

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>
```

Actor Languages Summary

- Actors are concurrent entities that react to messages.
 - State is completely encapsulated. There is no shared memory!
 - Message passing is asynchronous.
 - Actors can create new actors. 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.
- SALSA extends an object-oriented programming language (Java) with universal actors. State is explicit, encapsulated in instance variables. Control loop is implicit: ending a message handler, signals readiness to receive a new message. Actors are garbage-collected.
- 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. Ending a function denotes process (actor) termination.

Causal order

- In a sequential program all execution states are totally ordered
- In a concurrent program all execution states of a given actor are totally ordered
- The execution state of the concurrent program as a whole is partially ordered

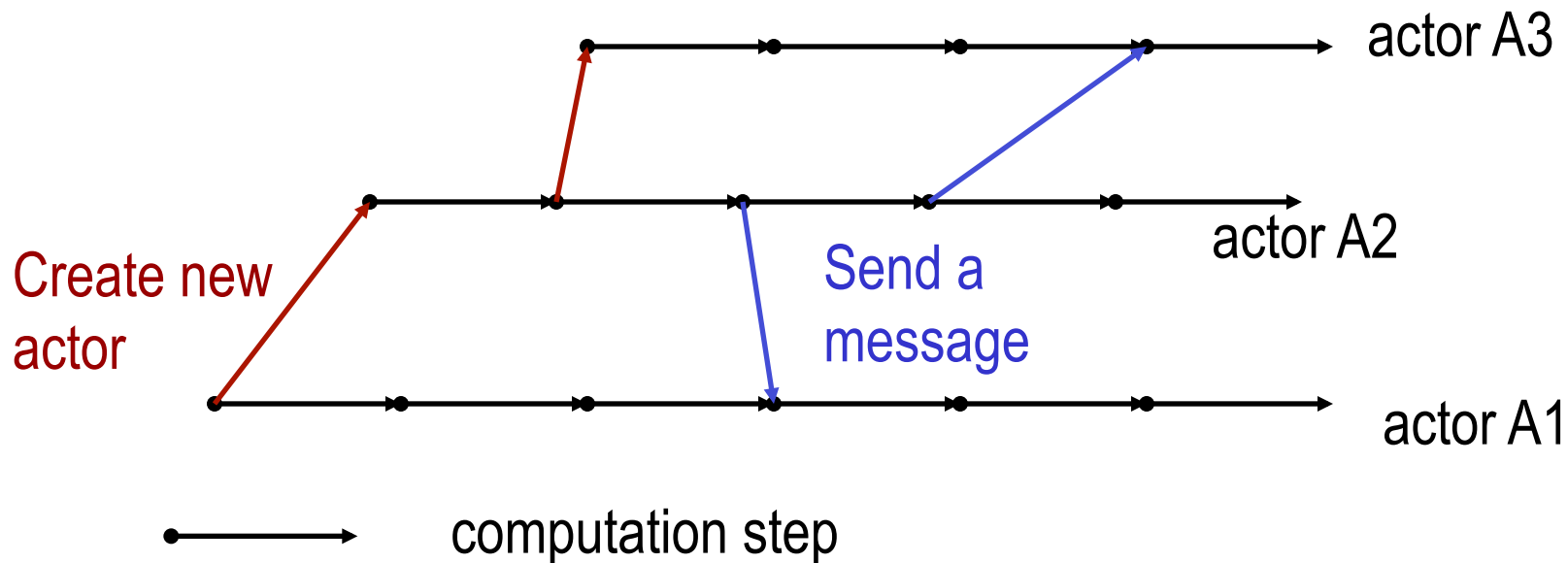
Total order

- In a sequential program all execution states are totally ordered



Causal order in the actor model

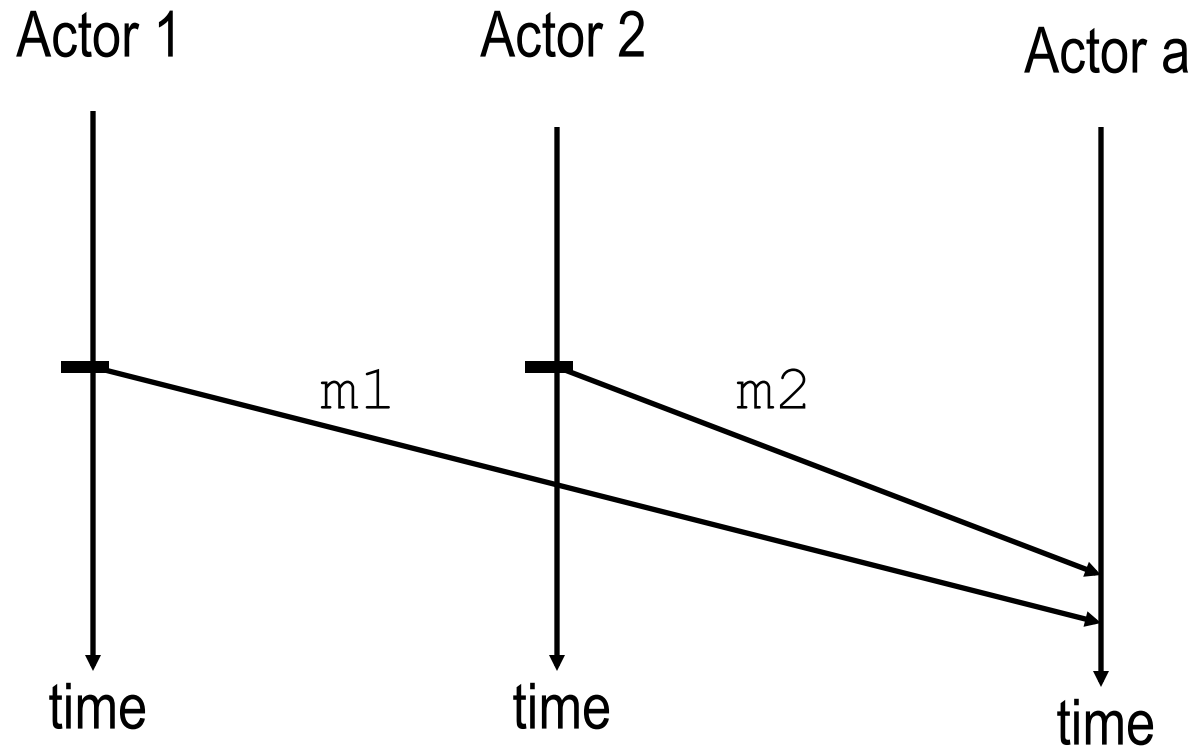
- In a concurrent program all execution states of a given actor are totally ordered
- The execution state of the concurrent program is partially ordered



Nondeterminism

- An execution is nondeterministic if there is a computation step in which there is a choice what to do next
- Nondeterminism appears naturally when there is asynchronous message passing
 - Messages can arrive or be processed in an order different from the sending order.

Example of nondeterminism



Actor a can receive messages $m1$ and $m2$ in any order.

Concurrency Control in SALSA

- SALSA provides three main coordination constructs:
 - Token-passing continuations
 - To synchronize concurrent activities
 - To notify completion of message processing
 - Named tokens enable arbitrary synchronization (data-flow)
 - Join blocks
 - Used for barrier synchronization for multiple concurrent activities
 - To obtain results from otherwise independent concurrent processes
 - First-class continuations
 - To delegate producing a result to a third-party actor

Token Passing Continuations

- Ensures that each message in the continuation expression is sent after the previous message has been **processed**. It also enables the use of a message handler return value as an argument for a later message (through the token keyword).

– Example:

```
a1 <- m1 () @  
a2 <- m2 ( token );
```

Send m1 to a1 asking a1 to forward the result of processing m1 to a2 (as the argument of message m2).

Token Passing Continuations

- @ syntax using token as an argument is syntactic sugar.

– Example 1:

```
a1 <- m1 () @  
a2 <- m2 ( token );
```

is syntactic sugar for:

```
token t = a1 <- m1 ();  
a2 <- m2 ( t );
```

– Example 2:

```
a1 <- m1 () @  
a2 <- m2 ();
```

is syntactic sugar for:

```
token t = a1 <- m1 ();  
a2 <- m2 () :waitfor ( t );
```

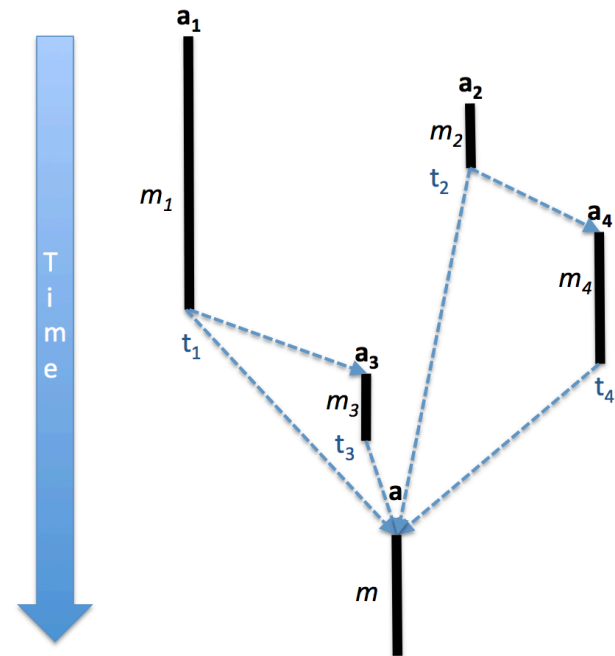
Named Tokens

- Tokens can be named to enable more loosely-coupled synchronization

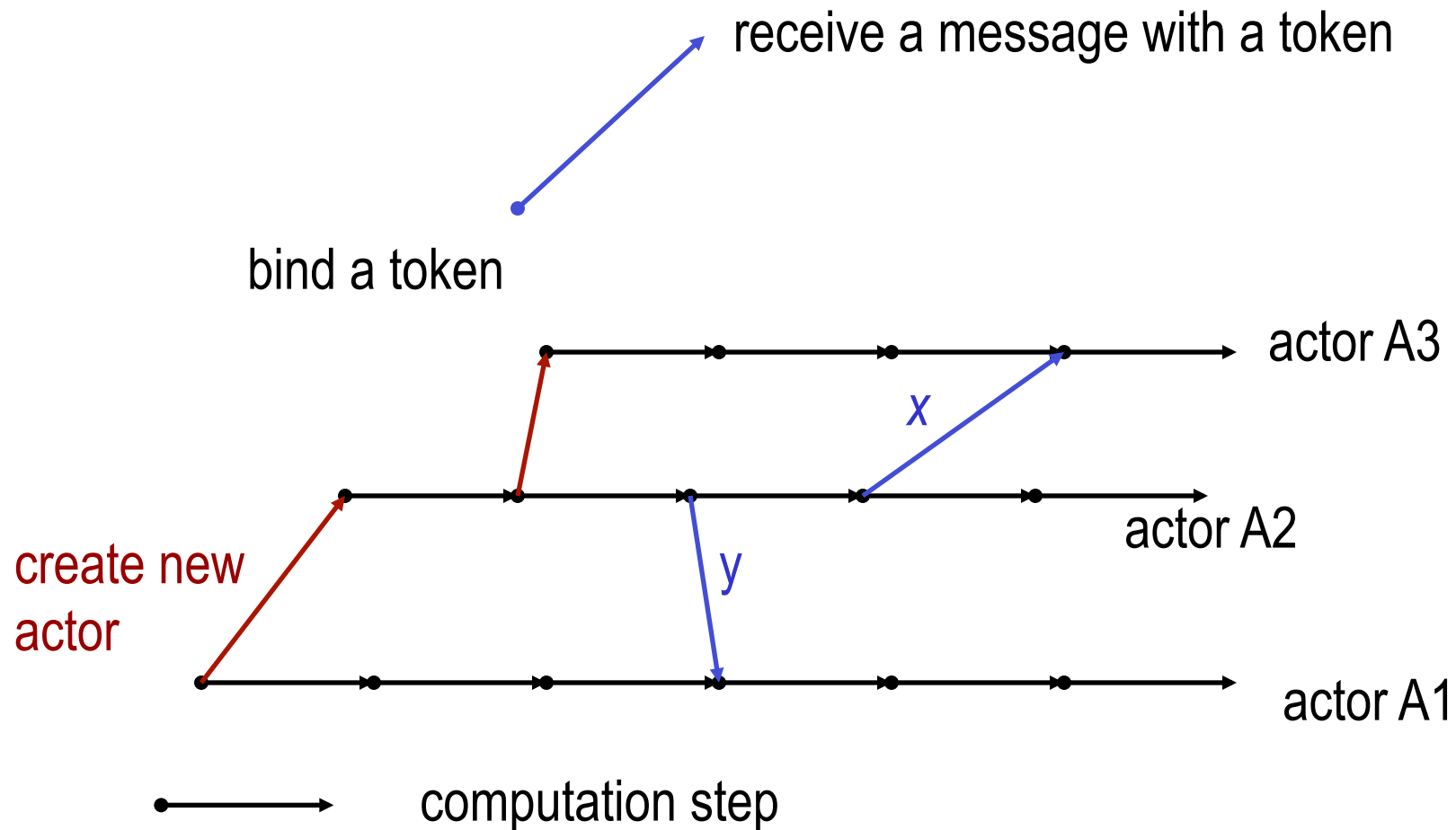
– Example:

```
token t1 = a1 <- m1 ();  
token t2 = a2 <- m2 ();  
token t3 = a3 <- m3 ( t1 );  
token t4 = a4 <- m4 ( t2 );  
a <- m ( t1, t2, t3, t4 );
```

Sending $m(\dots)$ to a will be delayed until messages $m1() \dots m4()$ have been processed. $m1()$ can proceed concurrently with $m2()$.



Causal order in the actor model



Deterministic Cell Tester Example

```
module cell;
```

```
behavior TokenCellTester {
```

```
  void act( String[] args ) {
```

```
    Cell c = new Cell(0);
```

```
    standardOutput <- print( "Initial Value:" ) @
```

```
    c <- get() @
```

```
    standardOutput <- println( token ) @
```

```
    c <- set(2) @
```

```
    standardOutput <- print( "New Value:" ) @
```

```
    c <- get() @
```

```
    standardOutput <- println( token );
```

```
  }
```

```
}
```

@ syntax enforces a sequential order of message execution.

token can be optionally used to get the return value (completion proof) of the previous message.

Cell Tester Example with Named Tokens

```
module cell;

behavior NamedTokenCellTester {

    void act(String args[]){

        Cell c = new Cell(0);
        token p0 = standardOutput <- print("Initial Value:");
        token t0 = c <- get();
        token p1 = standardOutput <- println(t0):waitfor(p0);
        token t1 = c <- set(2):waitfor(t0);
        token p2 = standardOutput <- print("New Value:"):waitfor(p1);
        token t2 = c <- get():waitfor(t1);
        standardOutput <- println(t2):waitfor(p2);

    }
}
```

We use p0, p1, p2 tokens to ensure printing in order.

We use t0, t1, t2 tokens to ensure cell messages are processed in order.

Join Blocks

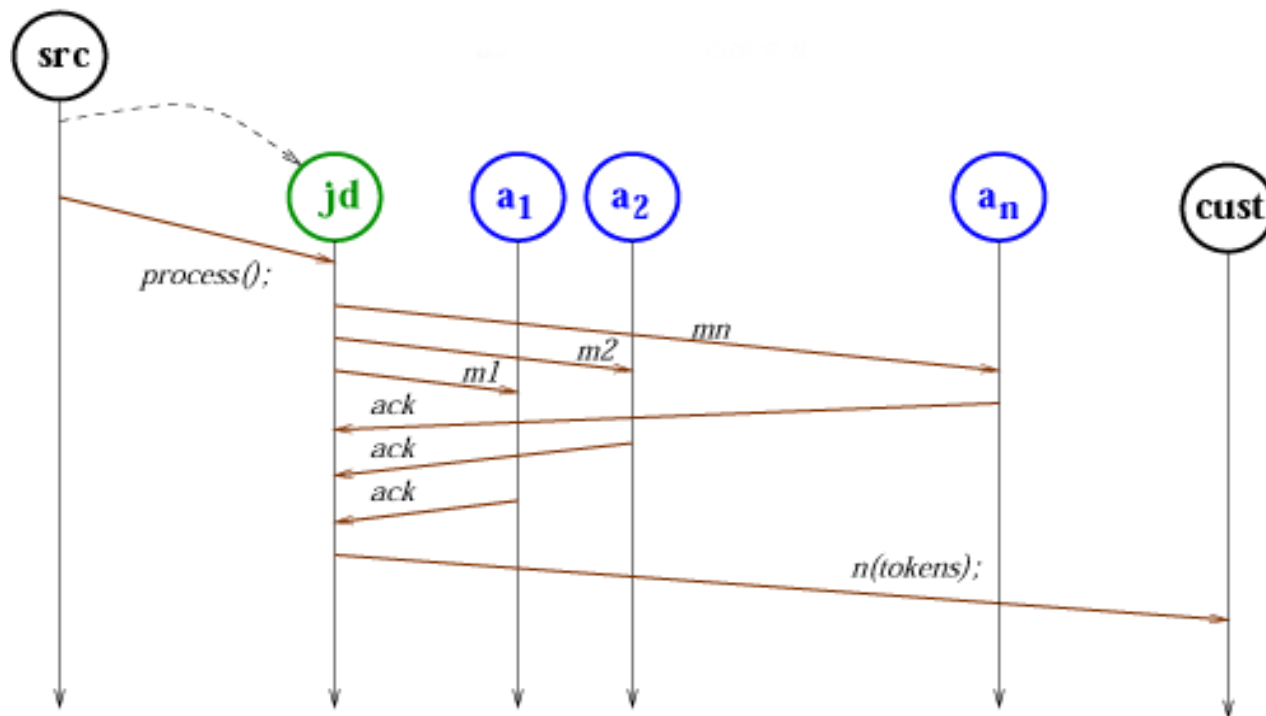
- Provide a mechanism for synchronizing the processing of a set of messages.
- Set of results is sent along as a *token* containing an array of results.
 - Example:

```
UniversalActor[] actors = { searcher0, searcher1,  
                             searcher2, searcher3 };  
  
join {  
    for (int i=0; i < actors.length; i++){  
        actors[i] <- find( phrase );  
    }  
} @ resultActor <- output( token );
```

Send the find(phrase) message to each actor in actors[] then after all have completed send the result to resultActor as the argument of an output(...) message.

Example: Acknowledged Multicast

```
join{ a1 <- m1(); a2 <- m2(); ... an <- mn(); } @  
cust <- n(token);
```



Lines of Code Comparison

	Java	Foundry	SALSA
Acknowledged Multicast	168	100	31

First Class Continuations

- Enable actors to delegate computation to a third party independently of the processing context.
- For example:

```
int m (...) {  
    b <- n (...) @ currentContinuation;  
}
```

Ask (delegate) actor b to respond to this message m on behalf of current actor ($self$) by processing b 's message n .

Delegate Example

```
module fibonacci;
```

```
behavior Calculator {
```

```
  int fib(int n) {  
    Fibonacci f = new Fibonacci(n);  
    f <- compute() @ currentContinuation;  
  }
```

```
  int add(int n1, int n2) {return n1+n2;}
```

```
  void act(String args[]) {  
    fib(15) @ standardOutput <- println(token);  
    fib(5) @ add(token,3) @  
    standardOutput <- println(token);  
  }  
}
```

fib(15)

is syntactic sugar for:

self <- fib(15)

Fibonacci Example

```
module fibonacci;

behavior Fibonacci {
    int n;

    Fibonacci(int n)          { this.n = n; }

    int add(int x, int y) { return x + y; }

    int compute() {
        if (n == 0)          return 0;
        else if (n <= 2)      return 1;
        else {
            Fibonacci fib1 = new Fibonacci(n-1);
            Fibonacci fib2 = new Fibonacci(n-2);
            token x = fib1<-compute();
            token y = fib2<-compute();
            add(x,y) @ currentContinuation;
        }
    }

    void act(String args[]) {
        n = Integer.parseInt(args[0]);
        compute() @ standardOutput<-println(token);
    }
}
```

Fibonacci Example 2

```
module fibonacci2;

behavior Fibonacci {

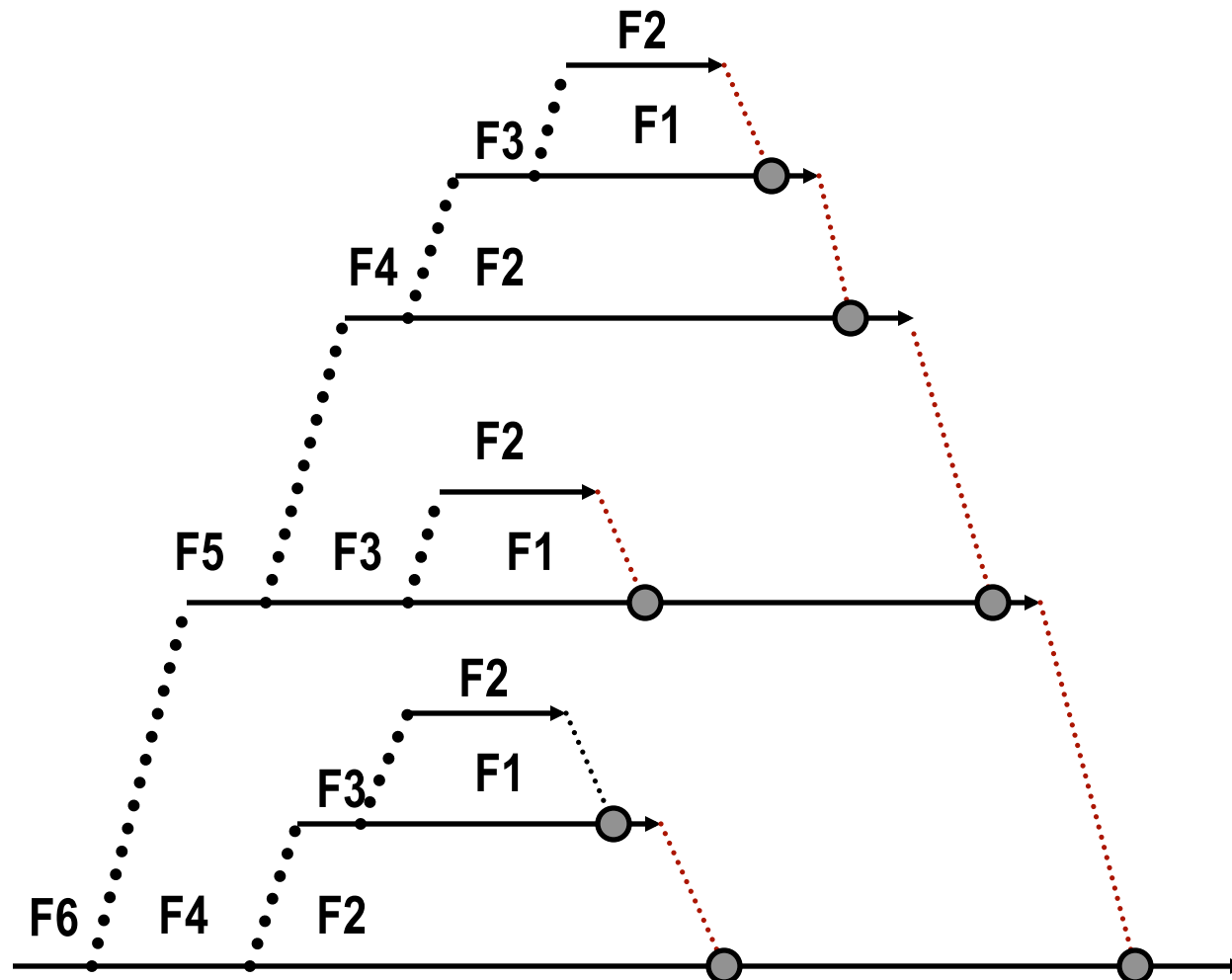
    int add(int x, int y) { return x + y; }

    int compute(int n) {
        if (n == 0) return 0;
        else if (n <= 2) return 1;
        else {
            Fibonacci fib = new Fibonacci();
            token x = fib <- compute(n-1);
            compute(n-2) @ add(x, token) @ currentContinuation;
        }
    }

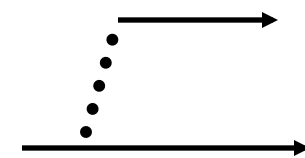
    void act(String args[]) {
        int n = Integer.parseInt(args[0]);
        compute(n) @ standardOutput<-println(token);
    }
}
```

compute(n-2) is a
message to self.

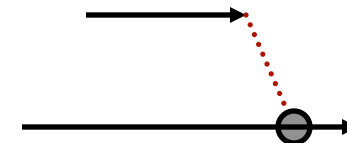
Execution of salsa Fibonacci 6



Create new actor



Synchronize on
result



Non-blocked actor



Tree Product Behavior Revisited

Notice we use token-passing continuations (@,token), a join block (join), and a first-class continuation (currentContinuation).

```
module treeprod;

behavior JoinTreeProduct {

    int multiply(Object[] results){
        return (Integer) results[0] * (Integer) results[1];
    }

    int compute(Tree t){
        if (t.isLeaf()) return t.value();
        else {
            JoinTreeProduct lp = new JoinTreeProduct();
            JoinTreeProduct rp = new JoinTreeProduct();
            join {
                lp <- compute(t.left());
                rp <- compute(t.right());
            } @ multiply(token) @ currentContinuation;
        }
    }
}
```

Concurrency control in Erlang

- Erlang uses a *selective receive* mechanism to help coordinate concurrent activities:
 - Message patterns and guards
 - To select the next message (from possibly many) to execute.
 - To receive messages from a specific process (actor).
 - To receive messages of a specific kind (pattern).
 - Timeouts
 - To enable default activities to fire in the absence of messages (following certain patterns).
 - To create timers.
 - Zero timeouts (`after 0`)
 - To implement priority messages, to flush a mailbox.

Selective Receive

```
receive
  MessagePattern1 [when Guard1] ->
    Actions1 ;
  MessagePattern2 [when Guard2] ->
    Actions2 ;
  ...
end
```

`receive` suspends until a message in the actor's mailbox matches any of the patterns including optional guards.

- Patterns are tried in order. On a match, the message is removed from the mailbox and the corresponding pattern's actions are executed.
- When a message does not match any of the patterns, it is left in the mailbox for future `receive` actions.

Selective Receive Example

Example program and mailbox (head at top):

```
receive
  msg_b -> ...
end
```

msg_a
msg_b
msg_c

`receive` tries to match `msg_a` and fails. `msg_b` can be matched, so it is processed. Suppose execution continues:

```
receive
  msg_c -> ...
  msg_a -> ...
end
```

msg_a
msg_c

The next message to be processed is `msg_a` since it is the next in the mailbox and it matches the 2nd pattern.

Receiving from a specific actor

```
Actor ! {self(), message}
```

`self()` is a Built-In-Function (BIF) that returns the current (executing) process id (actor name). Ids can be part of a message.

```
receive
  {ActorName, Msg} when ActorName == A1 ->
    ...
end
```

`receive` can then select only messages that come from a specific actor, in this example, A1. (Or other actors that know A1's actor name.)

Receiving a specific kind of message

```
counter(Val) ->
  receive
    increment -> counter(Val+1);
    {From,value} ->
      From ! {self(), Val},
      counter(Val);
    stop -> true;
    Other -> counter(Val)
  end.
```

increment is an atom
whereas **other** is a
variable (that matches
anything!).

`counter` is a behavior that can receive `increment` messages, `value` request messages, and `stop` messages. Other message kinds are ignored.

Order of message patterns matters

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

{Left, Right} is a more specific pattern than Number is (which matches anything!). Order of patterns is important.

In this example, a binary tree is represented as a tuple

{Left, Right}, or as a Number, e.g.,

{{{5, 6}, 2}, {3, 4}}

Selective Receive with Timeout

```
receive
  MessagePattern1 [when Guard1] ->
    Actions1 ;
  MessagePattern2 [when Guard2] ->
    Actions2 ;
  ...
  after TimeOutExpr ->
    ActionsT
end
```

`TimeOutExpr` evaluates to an integer interpreted as *milliseconds*.

If no message has been selected within this time, the timeout occurs and `ActionsT` are scheduled for evaluation.

A timeout of `infinity` means to wait indefinitely.

Timer Example

```
sleep(Time) ->  
  receive  
    after Time ->  
      true  
  end.
```

`sleep(Time)` suspends the current actor for `Time` milliseconds.

Timeout Example

```
receive
  click ->
    receive
      click ->
        double_click
      after double_click_interval() ->
        single_click
    end
  ...
end
```

`double_click_interval` evaluates to the number of milliseconds expected between two consecutive mouse clicks, for the receive to return a `double_click`. Otherwise, a `single_click` is returned.

Zero Timeout

```
receive
    MessagePattern1 [when Guard1] ->
        Actions1 ;
    MessagePattern2 [when Guard2] ->
        Actions2 ;
    ...
    after 0 ->
        ActionsT
end
```

A timeout of 0 means that the timeout will occur immediately, but Erlang tries all messages currently in the mailbox first.

Zero Timeout Example

```
flush_buffer() ->  
  receive  
    AnyMessage ->  
      flush_buffer()  
    after 0 ->  
      true  
  end.
```

`flush_buffer()` completely empties the mailbox of the current actor.

Priority Messages

```
priority_receive() ->  
  receive  
    interrupt ->  
      interrupt  
    after 0 ->  
      receive  
        AnyMessage ->  
          AnyMessage  
      end  
  end.  
end.
```

`priority_receive()` will return the first message in the actor's mailbox, except if there is an `interrupt` message, in which case, `interrupt` will be given priority.

Exercises

46. Download and execute the reference cell and tree product examples in SALSA and Erlang.
47. Write a solution to the Flavius Josephus problem in SALSA and Erlang. A description of the problem is at CTM Section 7.8.3 (page 558).
48. PDCS Exercise 9.6.6 (page 204).
49. How would you implement token-passing continuations, join blocks, and first-class continuations in Erlang?
50. How would you implement selective receive in SALSA?