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
Operational Semantics of Actors

\[ e \to \lambda e' \]
\[ \alpha, [R \triangleright e \triangleleft]_a \parallel \mu \quad \xrightarrow{[\text{fun}:a]} \quad \alpha, [R \triangleright e' \triangleleft]_a \parallel \mu \]

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

\[ \alpha' \text{ fresh} \]

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

\[ \alpha, [R \triangleright \text{ready}(b) \triangleleft]_a \parallel \{ \langle a \leftarrow v \rangle \} \uplus \mu \quad \xrightarrow{[\text{rcv}:a,v]} \quad \alpha, [b(v)]_a \parallel \mu \]
AMST Semantics Example

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

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

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

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

Multiple enabled transitions can lead to nondeterministic behavior.

The set of all computations sequences from \( k_0 \) is called the \textit{computation tree} \( \tau(k_0) \).
Actors/SALSA

• 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

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.
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);
    }
}
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.

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.
Tree Product Behavior in AMST

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

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

(a)

```
f(tree,cust)
```

(b)

```
f(left(tree),JC)
f(right(tree),JC)
```
Sample Execution

f(left(tree),JC)

(c)

(d)
Sample Execution
module jctreeprod;

import tree.Tree;

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);
    }
  }
}

C. Varela
Join Continuation in SALSA

module jctreeprod;

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.
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>
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

sequential execution

computation step
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.
Concurrence 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

C. Varela
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:

    ```language
    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:

```plaintext
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* \( m(\ldots) \) *to* \( a \) *will be delayed until messages* \( m1() \ldots m4() \) *have been processed. \( m1() \) *can proceed concurrently with \( m2() \).*
Causal order in the actor model

- Create new actor
- Bind a token
- Receive a message with a token

Actor A1 → Actor A2 → Actor A3

Computation step
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.
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 cell messages are processed in order.

We use t0, t1, t2 tokens to ensure printing 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:

```plaintext
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

\[
\text{join}\{ \text{a}_1 \leftarrow \text{m}_1(); \text{a}_2 \leftarrow \text{m}_2(); \ldots \text{a}_n \leftarrow \text{m}_n(); \} @ \\
\text{cust} \leftarrow \text{n(token)};
\]
## Lines of Code Comparison

<table>
<thead>
<tr>
<th>Acknowledged Multicast</th>
<th>Java</th>
<th>Foundry</th>
<th>SALSA</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>168</td>
<td>100</td>
<td>31</td>
</tr>
</tbody>
</table>
First Class Continuations

• Enable actors to delegate computation to a third party independently of the processing context.

• For example:

```c
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

```java
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)
```
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);
}

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

C. Varela
Tree Product Behavior Revisited

module treeprod;
import tree.Tree;

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;
        }
    }
}

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

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
```

The next message to be processed is msg_a since it is the next in the mailbox and it matches the 2\textsuperscript{nd} 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.)

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

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

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

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

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

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

Order of message patterns matters

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

C. Varela
sleep(Time) ->
    receive
        after Time ->
            true
        end.

sleep(Time) suspends the current actor for Time milliseconds.
Timeout Example

```prolog
receive
  click ->
  receive
    click ->
      double_click
    after double_click_interval() ->
      single_click
  end
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.
A timeout of 0 means that the timeout will occur immediately, but Erlang tries all messages currently in the mailbox first.
Zero Timeout Example

\[
\text{flush\_buffer}() \rightarrow \\
\text{receive} \\
\quad \text{AnyMessage} \rightarrow \\
\quad \quad \text{flush\_buffer}() \\
\quad \text{after 0} \rightarrow \\
\quad \quad \text{true} \\
\text{end.}
\]

\text{flush\_buffer}() \text{ completely empties the mailbox of the current actor.}
Priority Messages

```erlang
case receive
  when 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.
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?