CSCI.6500/4500 Distributed Computing over the Internet—Programming Distributed Computing Systems (Varela)—Sections 9.1, 9.3, 9.4

Instructor: Carlos Varela
Rensselaer Polytechnic Institute
Spring 2021
Concurrent programming with actors

In response to a message, an actor can:

1. modify its local state,
2. create new actors, and/or
3. send messages to acquaintances.
SALSA: Programming with Actors

The most significant semantic differences between SALSA and the actor language presented in Chapter 4, are the following:

- **Object-oriented core**
  - Agha, Mason, Smith and Talcott’s language (from now on, AMST) uses the $\lambda$ calculus to model sequential computation within an actor.
  - SALSA instead uses a sequential non-shared-memory subset of Java to model internal state and computation within an actor.
SALSA: Programming with Actors

• Classes as behaviors, methods as messages
  • AMST uses a lambda abstraction to model an actor’s behavior: receiving a message is modeled as applying the abstraction to the incoming message content.
  • SALSA uses classes (in object-oriented terms) to model actor behaviors:
    • individual actors are objects (instances of behavior classes) that conceptually encapsulate an independent thread of execution, and
    • messages are modeled as potential asynchronous method invocations on these instances.
      • Since method return values are produced asynchronously, we introduce tokens to represent the future values to be returned by messages.
# SALSA: Programming with Actors

<table>
<thead>
<tr>
<th>Actor-oriented programming</th>
<th>Object-oriented programming</th>
</tr>
</thead>
<tbody>
<tr>
<td>Behaviors</td>
<td>Classes (extending UniversalActor)</td>
</tr>
<tr>
<td>Actors</td>
<td>Objects (instances of behavior classes)</td>
</tr>
<tr>
<td>Messages</td>
<td>Asynchronous method invocations</td>
</tr>
<tr>
<td>Tokens</td>
<td>Message return values</td>
</tr>
</tbody>
</table>
SALSA: Programming with Actors

• **Static behaviors**
  
  • AMST enables actors to completely change their behavior when becoming *ready* to receive new messages.
  
  • SALSA’s actors always have the same static *behavior*, however, this behavior may depend on the internal state of the actor, which can change between message receptions.

• **Coordination constructs**
  
  • AMST uses asynchronous message sending as the only primitive form of communication.
  
  • SALSA provides a number of higher-level constructs that facilitate coordinating otherwise independent activities between actors.
SALSA: Programming with Actors

- Distribution and mobility
  - AMST does not explicitly model actor locations and mobility.
  - SALSA has a notion of *universal actor names* that enables
    - transparent actor migration and
    - location-independent communication.
## SALSA: Programming with Actors

A summary of the key AMST actor language primitives and the equivalent SALSA programming language syntax follows:

<table>
<thead>
<tr>
<th>AMST</th>
<th>SALSA</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>send(a, pr(m, v))</code></td>
<td><code>a &lt;- m(v);</code></td>
<td>Send named message</td>
</tr>
<tr>
<td><code>new(b)</code></td>
<td><code>new B();</code></td>
<td>Create new actor</td>
</tr>
<tr>
<td><code>ready(e)</code></td>
<td><code>s = e; return;</code></td>
<td>Behavior change</td>
</tr>
</tbody>
</table>
## SALSA: Programming with Actors

<table>
<thead>
<tr>
<th>AMST</th>
<th>SALSA</th>
<th>Send named message</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>send(a, pr(m, v))</code></td>
<td><code>a &lt;- m(v);</code></td>
<td></td>
</tr>
</tbody>
</table>

- AMST can use simple values as messages, on the other hand, SALSA actors use potential method invocations as messages.
  - Sending the message is asynchronous: the sender does not block but instead continues with its sequential computation.
  - The message is typically buffered, and when the recipient actor eventually receives the message, the corresponding method is actually executed.
### SALSA: Programming with Actors

<table>
<thead>
<tr>
<th>AMST</th>
<th>SALSA</th>
<th>Create new actor</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>new(b)</code></td>
<td><code>new B();</code></td>
<td>Create new actor</td>
</tr>
</tbody>
</table>

- In AMST, behaviors are $\lambda$ calculus abstractions, while in SALSA, the behavior is the name of a statically defined class.
Finally, in AMST an actor becomes ready to receive a new message with a new behavior $e$ by executing `ready(e)`.

In SALSA, internal state is modeled directly as internal variables which can be reassigned new values and persist across message receptions.

Returning from a method implicitly makes the actor ready to receive a new message.
Reference Cell in AMST and SALSA

\[
\text{cell} = \\
\quad \text{rec}(\lambda b.\lambda c.\lambda m. \\
\quad \quad \text{if}(\text{get}(m), \\
\quad \quad \quad \text{seq}(\text{send}(\text{cust}(m), c), \\
\quad \quad \quad \quad \text{ready}(b(c))), \\
\quad \quad \text{if}(\text{set}(m), \\
\quad \quad \quad \text{ready}(b(\text{contents}(m))), \\
\quad \quad \quad \text{ready}(b(c)))))
\]

behavior Cell{
    Object content;
    Cell(Object c){
        content = c;
    }
    Object get(){
        return content;
    }
    void set(Object c){
        content = c;
    }
}
A Ticker Example

Consider the AMST $ticker$ example:

$$ticker = \text{rec}(\lambda b. \lambda t. \lambda n. \lambda m. \text{seq}(\text{send}(t, \text{nil}), \text{ready}(b(t)(n + 1))))$$

We can write it in SALSA as follows:

```salma
behavior Ticker{
    int n;
    Ticker(int n){
        this.n = n;
    }
    void tick(){
        n++;
        self<-tick();
    }
}
```
A Ticker Example

The Ticker behavior has

- a state variable, n;

- a constructor, which following Java’s convention uses the same name as the behavior; and

- a message handler tick that changes the actor’s internal state and sends itself a message to continue ticking.

Notice that we do not need to explicitly keep an actor’s own name, since the keyword self represents it.

- In fact, when a message has no explicit recipient, it is sent to self.

- So, self<-tick(); and simply tick(); are equivalent.

Also notice that we can use the this keyword to refer to the actor’s state variables as opposed to the constructor/method formal arguments.
A Ticker Example

In AMST, it is created and started as follows:

\[
\text{letrec } t = \text{new}(ticker(t)(0)) \text{ in send}(t, \text{nil})
\]

Equivalent code in SALSA follows:

```plaintext
Ticker ticker = new Ticker(0);
ticker <- tick();
```
SALSA Programming Language Syntax

\( A_v = \{ \) null } \)  \hspace{1cm} Void  \\
\( A_b = \{ \) true, false} \)  \hspace{1cm} Booleans  \\
\( A_i = \{ ..., -2, -1, 0, 1, 2, ... \} \)  \hspace{1cm} Integers  \\
\( A ::= A_v \mid A_b \mid A_i \mid ... \)  \hspace{1cm} Atoms  \\
\( B_p = \{ \) void, boolean, int, char, ...} \)  \hspace{1cm} Primitive types  \\
\( B_j = \{ \) Object, Integer, String, ...} \)  \hspace{1cm} Java class names  \\
\( B_u = \{ A, B, C, ... \} \)  \hspace{1cm} User-defined behaviors  \\
\( B ::= B_p \mid B_j \mid B_u \)  \hspace{1cm} Behavior names  \\
\( X_k = \{ \) self, token, this, super, standardOutput, ...} \)  \hspace{1cm} Key words  \\
\( X_u = \{ x, y, z, ... \} \)  \hspace{1cm} User-defined variables  \\
\( X ::= X_k \mid X_u \)  \hspace{1cm} Variables  \\
\( F = \{ +, *, ==, >, ... \} \)  \hspace{1cm} Primitive operators  \\
\( D = \{ a, b, c, ... \} \)  \hspace{1cm} Actor names  \\
\( T = \{ t, t_0, t_1, ... \} \)  \hspace{1cm} Tokens  \\
\( U = \{ u, u_0, u_1, ... \} \)  \hspace{1cm} Universal actor names  \\
\( L = \{ s, s_0, s_1, ... \} \)  \hspace{1cm} Universal actor locations  \\
\( M = \{ migrate, m, n, o, ... \} \)  \hspace{1cm} Message names  \\
\( M_p = \{ waitfor, priority, ... \} \)  \hspace{1cm} Message properties
SALSA Programming Language Syntax

\[ V ::= A | \mathcal{X} | D | T | U | L \]

\[ E ::= V | \mathcal{F}(E, \ldots, E) | E \mathcal{F} E | (E) | E.\mathcal{X} | E.M(E, \ldots, E) | \text{new } B(E, \ldots, E) \begin{array}{c} \text{at } (U, L) \end{array} | \text{reference } U \]

\[ S_d ::= B \mathcal{X} \begin{array}{c} [ = E ] \end{array} \]

\[ S_p ::= E <- M(E, \ldots, E) | \text{join } \{ S; \ldots; S \} \]

\[ S_m ::= \begin{array}{c} \begin{array}{c} [\text{token } \tau =] \end{array} \end{array} S_p \begin{array}{c} [:: M_p(E, \ldots, E) :: \ldots :: M_p(E, \ldots, E)] \end{array} \]
SALSA Programming Language Syntax

\[ S_c ::= \]
\[ \text{@ currentContinuation} \]
\[ \text{@ } S_m[S_c] \]

\[ S ::= \]
\[ S_d \]
\[ \{ S; \ldots; S \} \]
\[ [E.]X = E \]
\[ S_m[S_c]; \]
\[ \text{if} (E) S \text{ else } S \]
\[ \text{while} (E) S \]
\[ \text{for} ([S];[E];[S]) S \]
\[ \text{return} [E]; \]

\[ H ::= B \ M(B X, \ldots, B X) \{ S; \ldots; S \} \]

\[ C ::= B(B X \ldots B X) \{ S; \ldots; S \} \]

\[ P ::= \text{behavior } B\{S_d; \ldots; S_d; C \ldots C H \ldots H\} \]
Hello World Example

```java
module helloworld;

behavior HelloWorld {
    void act(String arguments[]){
        standardOutput<-print("Hello");
        standardOutput<-println(" World!");
    }
}
```

This code can print “Hello World!” or “ World!Hello”, since message passing is asynchronous.
Hello World Example

- The `act(String[] args)` message handler is similar to the `main(...)` method in Java and is used to bootstrap SALSA programs.

- When a SALSA program is executed, an actor of the given behavior is created and an `act(args)` message is sent to this actor with any given command-line arguments.

- References to `standardOutput`, `standardInput` and `standardError` actors are available to all SALSA actors.
Tree Product Behavior in AMST

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

\[ \text{joincont} = \lambda \text{customer}.\lambda \text{firstnum}. \]
\[ \text{ready}(\lambda \text{num}. \text{seq}(\text{send}(\text{customer}, \text{firstnum} \times \text{num}) , \]
\[ \text{ready}(\text{sink}))) \]
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);
        }
    }
}
Tree Product Behavior 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);
    }
}
Causality in Actor Systems

Activities in actor systems happen in response to messages. These activities are typically independent from each other and can happen concurrently and in any order. Causality conditions constrain these otherwise independent activities to occur in a partial order. Actor systems have only two causality conditions:

- **Actor creation** If activity $p$ in actor $a$ preceeds the creation of actor $b$, then activity $p$ must preceed every activity $q$ in actor $b$.

- **Message sending** If activity $q$ in actor $b$ happens in response to a message $m$ from actor $a$, then all activities that preceed the message sending from $a$ also must preceed $q$. 
Causality in Actor Systems

Asynchronous messaging and state encapsulation are important properties for systems modularity:

• they enable easier distribution and dynamic reconfiguration of software sub-components, which in turn
• promote scalability and fault tolerance.

However, programming only with pure asynchronous messaging is very low level and prone to error.

In the SALSA programming language, there are higher level coordination constructs that enable building more complex interaction patterns, without sacrificing modularity.
SALSA Coordination Abstractions

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
Continuations

Continuations have been used in functional programming languages as a way to tell a function how to proceed after the function is computed. For example, there may be

- a *success* continuation specifying how to continue the computation if no errors are encountered and
- a *failure* continuation specifying how to handle anomalies, such as division by zero.
Token-passing Continuations

Token-passing continuations in SALSA also enable to establish causality conditions among otherwise independent activities.

- Since actor messages in SALSA are modeled as potential method invocations, it is natural to want to use the result of invoking a method, even if this invocation will happen asynchronously and in the future.

- We call the result of a future message invocation, a token, and we allow the use of the token only as an argument to future messages.
Token-passing Continuations

Consider, for example, the code:

```c
checking <- getBalance() @
savings <- transfer(checking, token);
```

In this example, we send a `getBalance` message to a checking account actor, and we specify the continuation as sending the message `transfer` to the savings account actor with two parameters:

- first, the checking account actor name, and
- second, the token that represents the return value of the `getBalance` message to the checking account actor.

This example could also have been written as:

```c
token balance = checking <- getBalance();
savings <- transfer(checking, balance);
```
module cell;

behavior IntCellTester {

    void act( String[] args ) {

        Cell c = new Cell(0);
        c <- set(7);
        c <- get() @
        standardOutput <- println(token);

    }

}
Named Tokens

Named tokens enable arbitrary data-flow computations to be specified. For example, consider the following code:

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

• Actor a’s processing of message m will be delayed until messages m1, ..., m4 have been processed.

• m4 can proceed concurrently with m1 and m3.

• m3 can also proceed concurrently with m2 and m4, but m3 can only happen after m1 has been processed and likewise,

• m4 can only happen after m2 has been processed.
module cell;

behavior IntCellTester {

    void act( String[] args ) {

        Cell c = new Cell(0);
        c <- set(7);
        token t = c <- get();
        standardOutput <- println(t);
    }
}

Deterministic Hello World

```haskell
module helloworld;

behavior HelloWorld {
    void act(String arguments[]){
        standardOutput<-print("Hello") @
        standardOutput<-println(" World!");
    }
}
```

This code can only print “Hello World!” since a token-passing continuation is used—an empty (void) token is used for synchronization.
Join Blocks

*Join blocks* in SALSA enable to program *rendezvous*-like interaction patterns.

- Several independent activities are barrier-synchronized so that
- only after all of them have finished execution does a given join continuation become enabled.
Join Blocks

For example, consider the following code:

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

- This code sends the `find( phrase )` message to each actor in the `actors` array and after all the searcher actors have processed their respective messages, the `output` message is sent to the `customer` actor.
- The set of individual results (or tokens) is combined as a single array of tokens and sent as an argument to the `output` message.
Deterministic String Cell Tester

module cell;
behavior DCellTester{

    void print(Object[] args){
        standardOutput<-println(args[1]);
    }

    void act(String args[]){
        Cell c = new Cell("Hello");
        join {
            standardOutput<-print("Initial Value:");
            c<-get();
        } @ print(token)
        join {
            standardOutput<-print("New Value:");
            c<-set("World") @ c<-get();
        } @
        print(token);
    }
}

CSCI.6500/4500 Distributed Computing over the Internet—Programming Distributed Computing Systems (Varela)—Sections 9.1,9.3,9.4 – p. 36/44
Delegation through First-class Continuations

First-class continuations enable actors to delegate computation to a third party independently of the current processing context. For example, consider the following code fragment:

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

An instance `a` of this behavior `A`, when processing message `m` asks (delegates) actor `b` to respond to this message `m` on its behalf by processing its message `n`. In this example, actor `b`’s message handler for `n` should return a value of type `int`. 
Delegation through First-class Continuations

Let us consider an example:

```java
behavior Calculator {
    int fib(int n) {
        Fibonacci f = new Fibonacci();
        f <- compute(n) @ 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);
    }
}
```
Delegation through First-class Continuations

The Calculator behavior delegates the computation of Fibonacci numbers (\texttt{fib}) to a newly created Fibonacci actor.

- In response to the \texttt{act} message, the calculator actor sends itself two messages.
- The first one requests the calculation of \texttt{fib(15)} with the continuation specified as printing the result to standard output.
- The second request (\texttt{fib(5)}) specifies a different continuation:
  - to add the result to the number 3 by sending itself an \texttt{add} message with the token (result of \texttt{fib(5)}) and the number 3 as arguments, and then
  - send the result to the \texttt{standardOutput} actor for printing.
The Fibonacci behavior could be encoded as follows:

```java
behavior Fibonacci {
    int compute(int n) {
        if (n == 0) return 0;
        else if (n <= 2) return 1;
        else {
            Fibonacci fib = new Fibonacci();
            Calculator calc = new Calculator();
            token x = fib <- compute(n-1);
            compute(n-2) @
            calc <- add(x,token) @
            currentContinuation;
        }
    }
}
```
Delegation through First-class Continuations

Notice that the Fibonacci behavior creates two helper actors to perform the computation:

- an actor with the Fibonacci behavior to compute \( \text{fib}(n-1) \) and
- an actor with the Calculator behavior to compute the addition of \( \text{fib}(n-1) \) (produced by the helper actor) and \( \text{fib}(n-2) \) (produced by itself.)

Notice how the currentContinuation keyword enables passing the current continuation, which is different at the top level and at each recursive step.
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 Revisited

The TreeProduct example combines:

- a *token*-passing continuation: \@ \texttt{multiply(token)},
- a *join* block: to join the left and right sub-tree product computations, and
- a *first-class* continuation: \@ \texttt{currentContinuation} to delegate the return of \texttt{compute} to the \texttt{multiply} message handler.

It also uses a Java class (\texttt{tree.Tree}) to represent a binary tree.
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);
    }
}