Concurrency control abstractions in SALSA (PDCS 9.2.4)

Carlos Varela

Rennselaer Polytechnic Institute

October 15, 2024

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 in AMST

$B_{treeprod}$ =

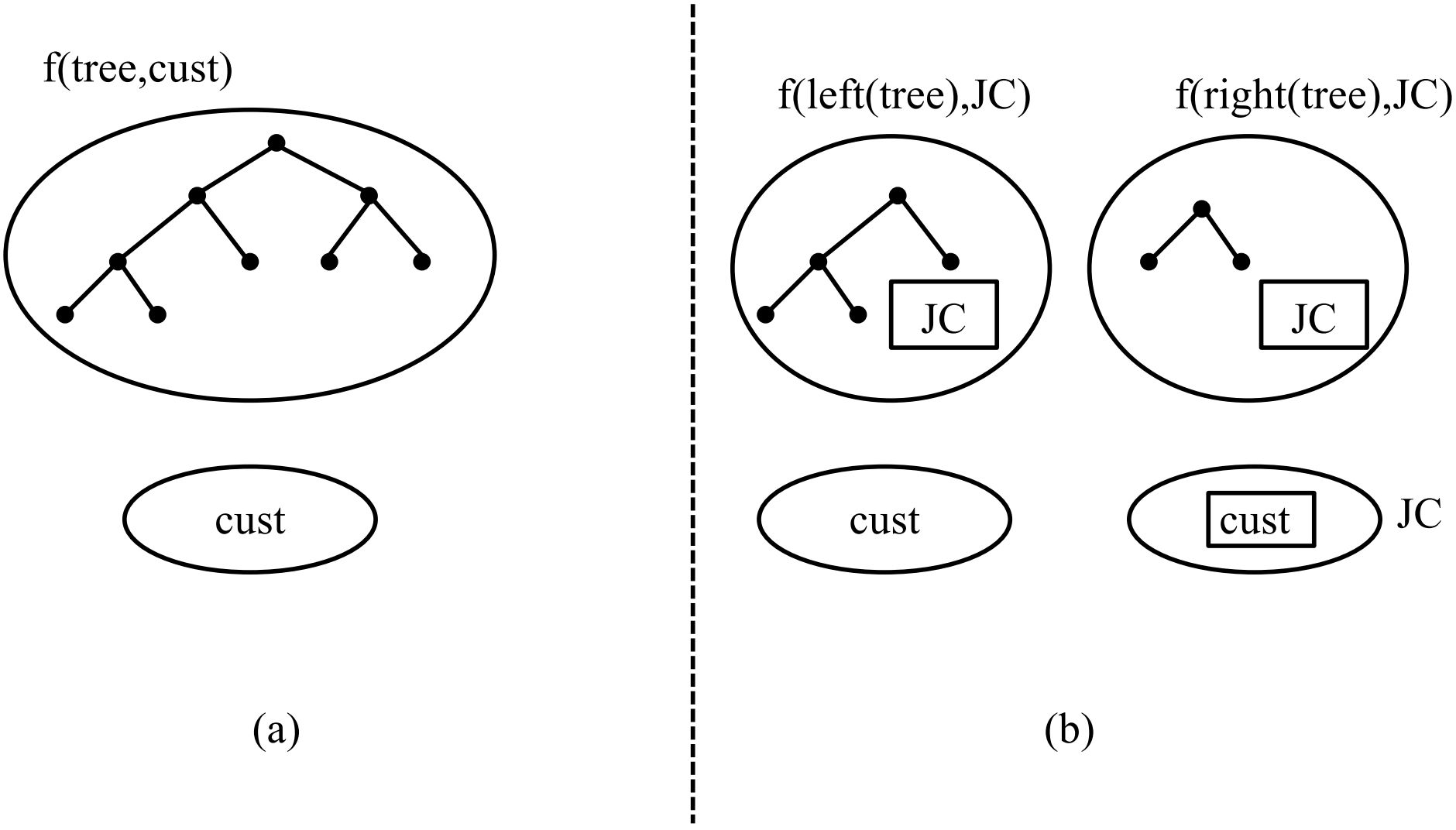
```
   rec(λb.λm.
       seq(if(isnat(tree(m)),
              send(cust(m),tree(m)),
              let newcust=new(B_joincont(cust(m))),
                   lp = new(b),
                   rp = new(b) in
              seq(send(lp,
                   pr(left(tree(m)),newcust)),
                  send(rp,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

Join Continuation in AMST

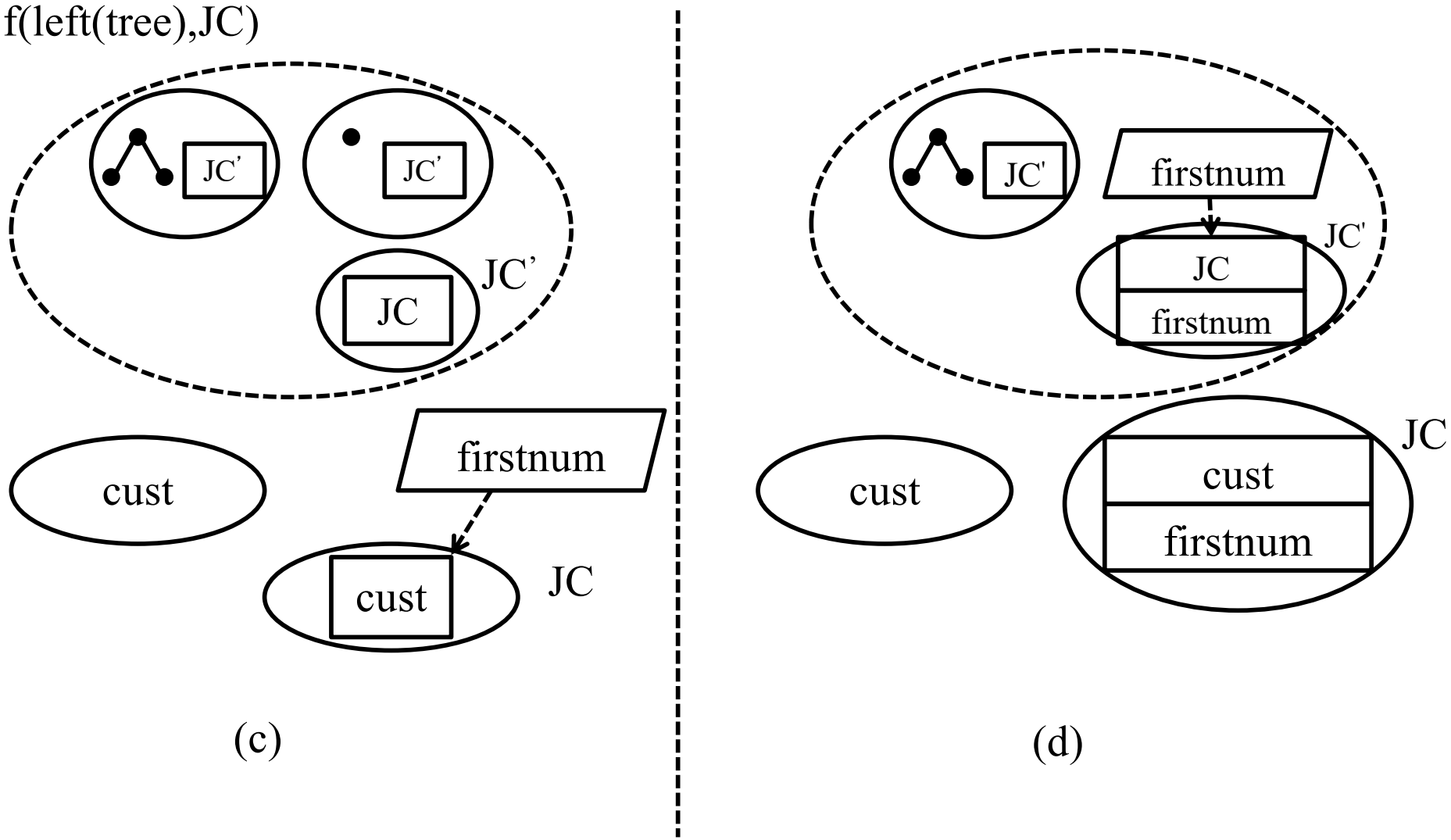
$B_{joincont}$ =

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

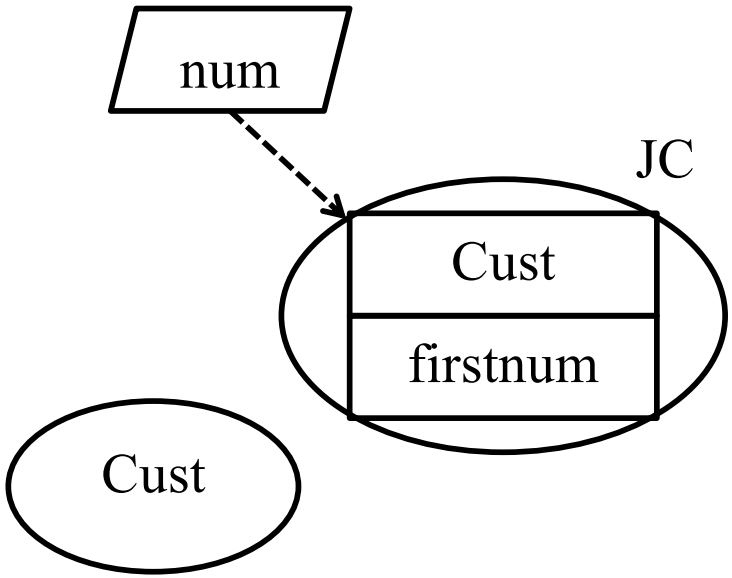
Sample Execution



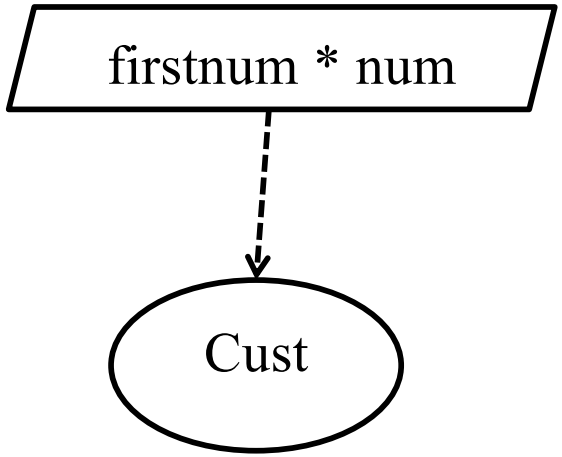
Sample Execution



Sample Execution



(e)



(f)

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>
```

Tree Product Behavior in SALSA

This code uses token-passing continuations (`@,token`), a join block (`join`), and a first-class continuation (`currentContinuation`).

```
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 Tester

```
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);
    }
}
```

```
Use as follows:
% javac tree/Tree.java
% salsac treeprod/*
% salsa treeprod/TreeProductTester
720
```

Actor Languages Summary

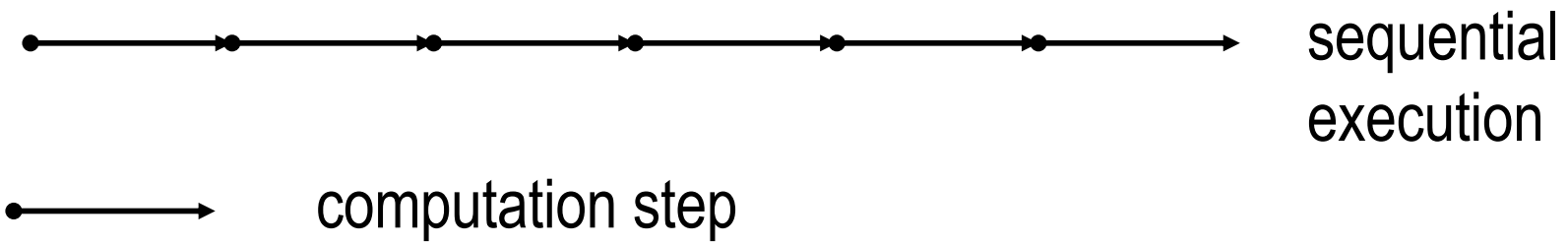
- Actors are concurrent entities that react to messages.
  - State is completely encapsulated. There is no shared memory!
  - Message passing is asynchronous.
  - Actors can create new actors. Run-time has to ensure fairness.
- AMST extends the call by value lambda calculus with actor primitives. State is modeled as function arguments. Actors use `ready` to receive new messages.
- 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.
- 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.

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

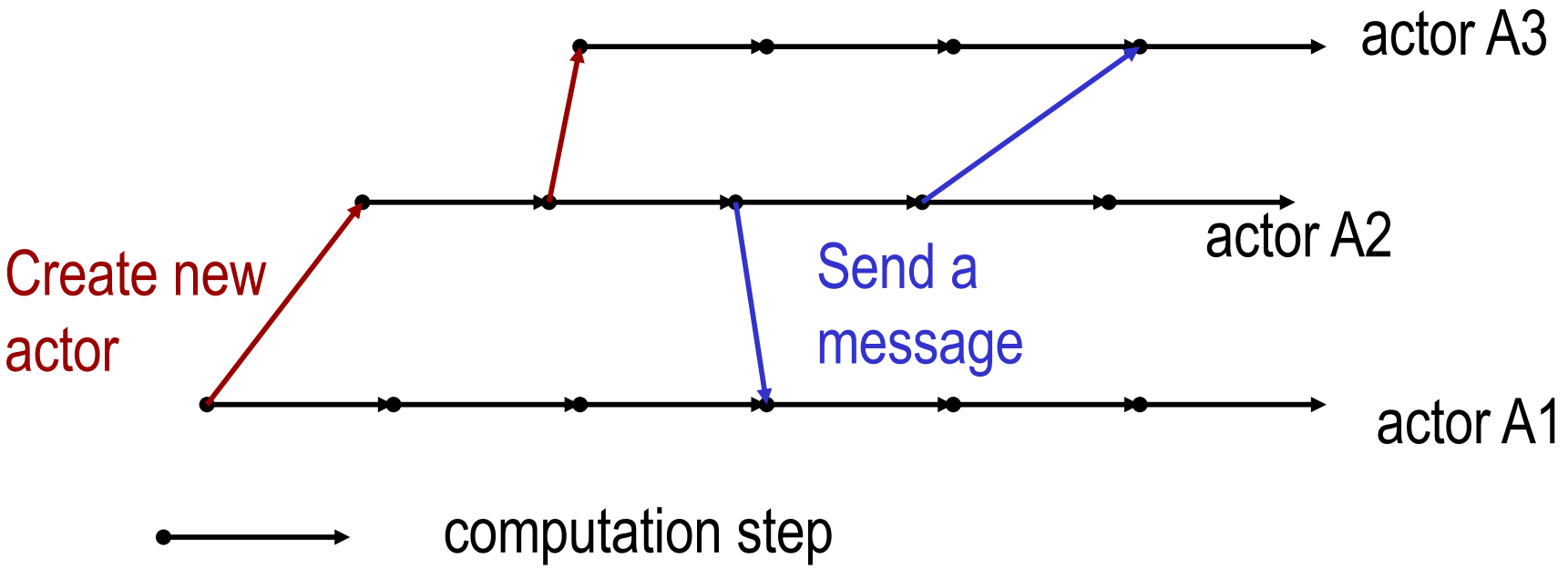
Total order

- 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

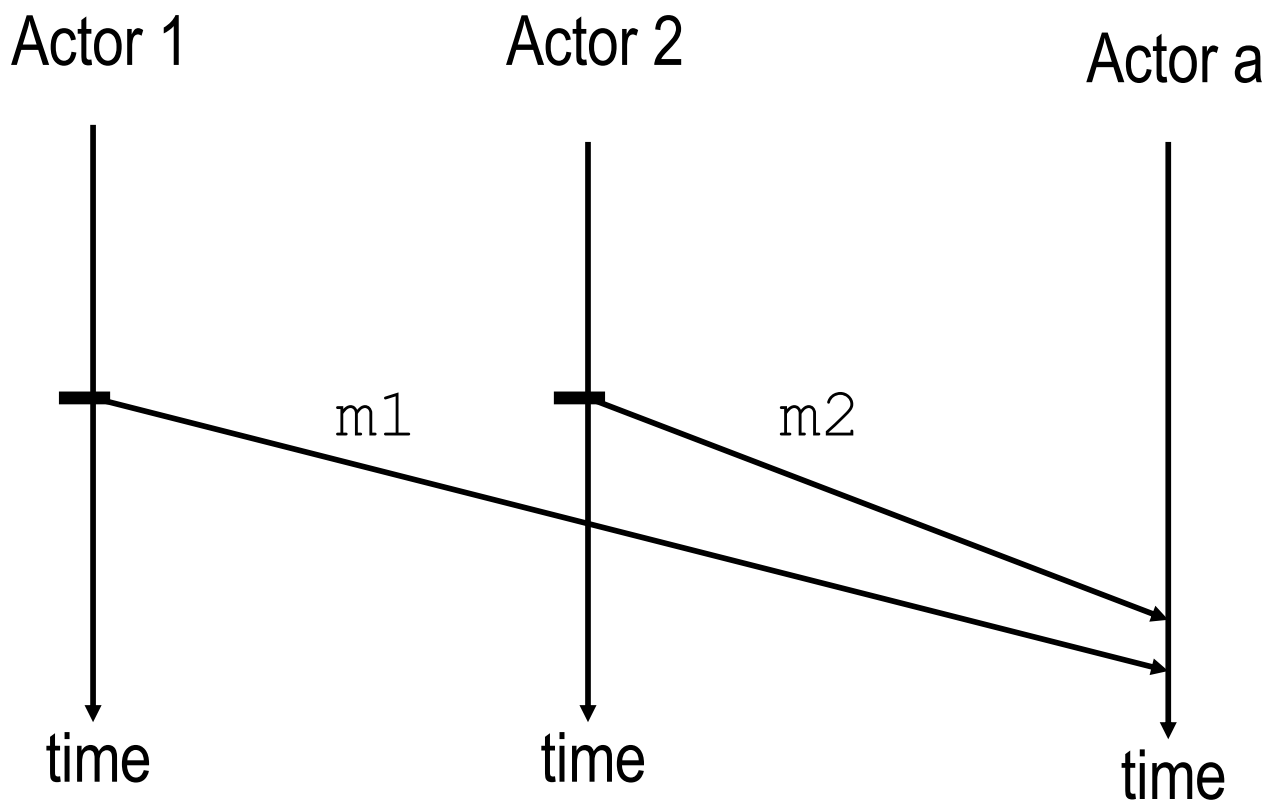


Nondeterminism

- 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.

Tree Product Behavior Revisited

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

```
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;
      }
    }
}
```

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 another message, or 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:

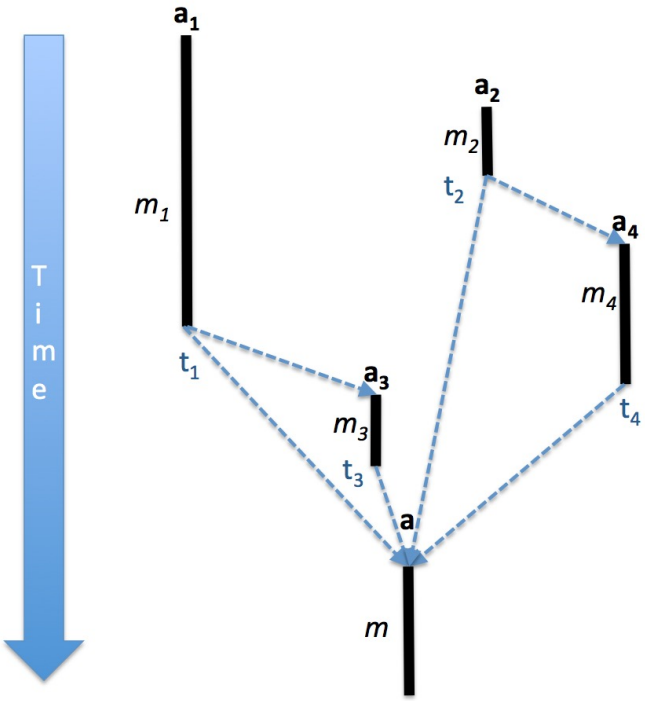
```
token t = a1 <- m1();
a2 <- m2():waitfor( t );
```

Named Tokens

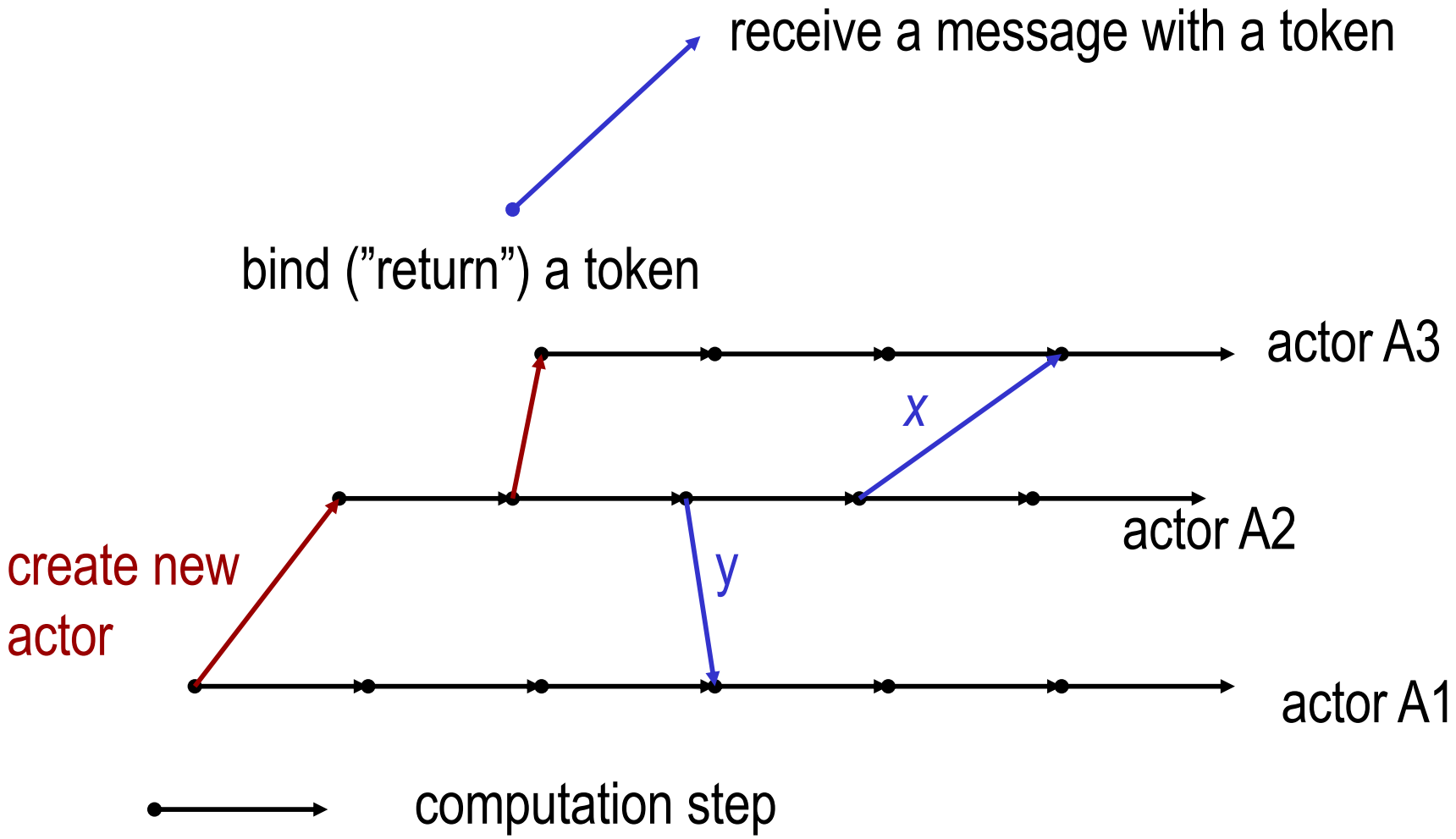
- Tokens can be named to enable more loosely-coupled synchronization
  - Example:

```
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(...)` to `a` will be delayed until messages `m1()..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

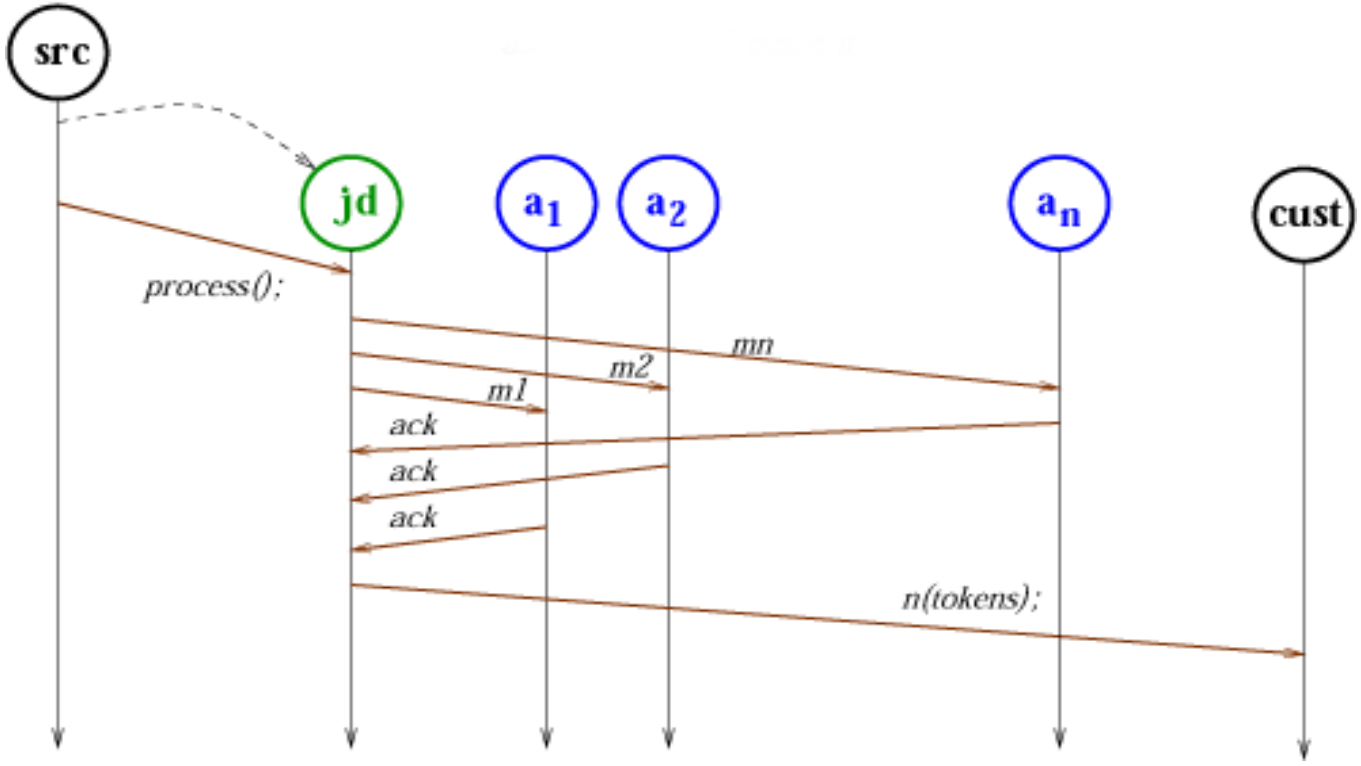
- Provide a mechanism for synchronizing the processing of a set of messages.
- Set of results is sent along as a *token* containing an array of results.
  - Example:

```
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

```
fib(15)
```

is syntactic sugar for:

```
self <- fib(15)
```

```
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);
   }
}
```

Fibonacci Example

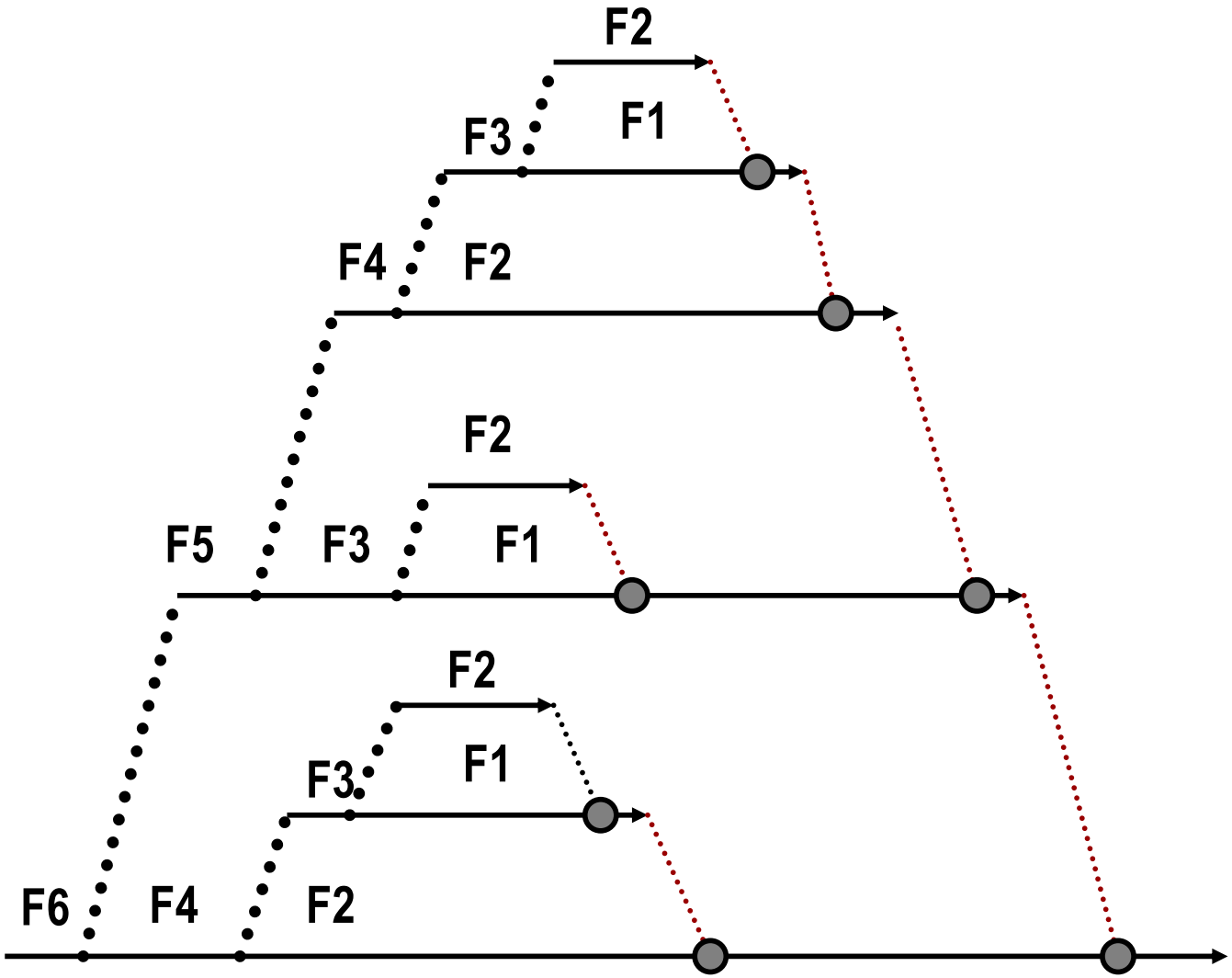
```
module fibonacci;

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);
    }
}
```

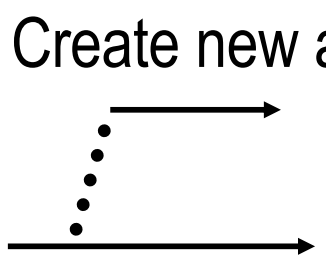
Fibonacci Example 2

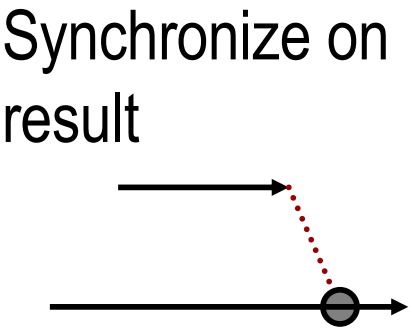
```
module fibonacci2;

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);
    }
}
```

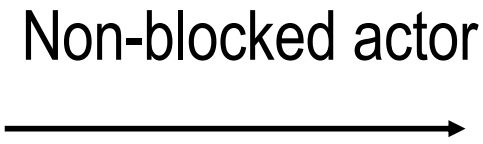
compute(n-2) is a message to self.

Execution of `salsa Fibonacci 6`









Exercises

45. Download and execute the tree product example in SALSA.
46. Modify the `treeprod` behavior in Erlang to reuse the tree product actor to compute the product of the left subtree. (See PDCS page 63 for the corresponding $\texttt{tprod}_2$ behavior in AMST.)
47. Create a concurrent `fibonacci` behavior in SALSA using a join block.
48. Write a solution to the Flavius Josephus problem in SALSA. A description of the problem is at CTM Section 7.8.3 (page 558).
49. PDCS Exercise 9.6.6 (page 204).