LINFO1104
Concepts, paradigms, and semantics of programming languages

Lecture 10-11

Peter Van Roy

ICTEAM Institute
Université catholique de Louvain

peter.vanroy@uclouvain.be

1

Overview of lectures 10-11

- Limitation of deterministic dataflow
  - Some applications cannot be written in deterministic dataflow! We explain why not.
- Message-passing concurrency (multi-agent actor programming)
  - We overcome the limitation by adding a new concept, ports, to deterministic dataflow
  - We define port objects and active objects
  - We show how to write message protocols and large multi-agent actor programs

2

Limitation of deterministic dataflow

3

Limitation of deterministic dataflow

- In lectures 8-9 we saw deterministic dataflow, which makes concurrent programming very easy
  - It allows “Concurrency for Dummies”: threads can be added to the program at will without changing the result
- But unfortunately it cannot be used all the time!
  - It has a strong limitation: it cannot be used to write programs when the nondeterminism must be visible
  - But why must nondeterminism sometimes be visible? Let’s see an example: a client/server application.

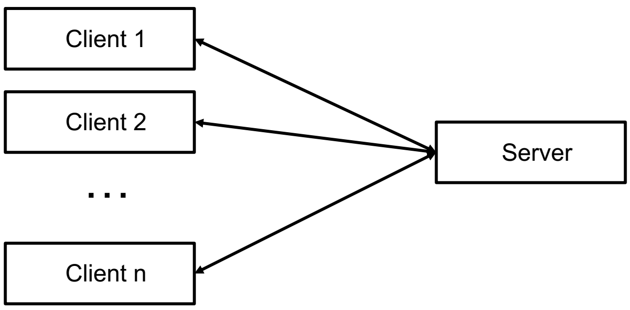
4

Client/server application (1)

- A client/server application consists of a set of clients all communicating to one server
  - The clients and the server are concurrent agents
  - Each client sends messages to the server and receives replies
- Client/server applications are ubiquitous on the Internet
  - For example, all Web stores are client/servers: the users are clients and the store is the server
  - When shopping at Amazon, your Web browser sends messages and receives replies from the Amazon server
- Client/server cannot be written in deterministic dataflow!
  - Why not? Let's try and see what goes wrong! Try it yourself!

5

Client/server application (2)



- Each client has a link to the server and can send messages to the server at any time
- The server receives each message, does a local computation, and then replies immediately

6

Client/server: first attempt

- Let's try to write a client/server in deterministic dataflow
  - Assume that there are two clients, each with an output stream, and the server receives both
- Here is the server code:

This doesn't work!
Explain why not.

```
proc {Server S1 S2}
    case S1|S2 of (M1|T1)|S2 then
            (handle M1) {Server T1 S2}
    [] S1|(M2|T2) then
            (handle M2) {Server S1 T2}
    end
end
```

7

Client/server: second attempt

- The first attempt does not work if Client 2 sends a message and Client 1 sends nothing
- We can try doing it the other way around:

```
proc {Server S1 S2}
    case S1|S2 of S1|(M2|T2) then
            (handle M1) {Server S1 T2}
    [] (M1|T1)|S2 then
            (handle M2) {Server T1 S2}
    end
end
```

- This doesn't work if Client 1 sends a message and Client 2 sends nothing!

8

Client/server: third attempt

- Maybe the server has to receive from both clients:

```
proc {Server S1 S2}
   case S1|S2 of (M1|T1)|(M2|T2) then
         (handle M1)
         (handle M2) {Server T1 T2}
   end
end
```

- This does not work either! (Why not?)

9

What is the problem?

- The **case** statement waits on a single pattern
  - This is because of determinism: with the same input, the **case** statement must give the same result
- But the server must wait on two patterns
  - Either M1 from Client 1 or M2 from Client 2
  - Either pattern is possible, it depends on when each client sends the message and on how long the message takes to reach the server
    - The decision is made outside the program
  - This means exactly that execution is nondeterministic!

10

Understanding nondeterminism

- Nondeterminism means that a choice is made outside of the program's control
  - This is exactly what is happening here: the choice is the arrival order of the client messages, which depends on the human clients and on the message travel time
- The nondeterminism is inherently part of the client/server execution, it cannot be avoided
  - The nondeterminism is a consequence of the initial requirement: "The server receives each message, does a local computation, and then replies immediately"
  - This means that the reply cannot be delayed while the server waits for another message

11

Overcoming the limitation

12

Overcoming the limitation

- Deterministic dataflow cannot express an application that requires nondeterminism
- To do this, we need to extend the kernel language with a new concept
- The new concept must be able to wait on several events nondeterministically
  - The new language is no longer deterministic!
- We will show two possible solutions

13

Solution 1: WaitTwo

- We introduce the function:
  {WaitTwo X Y}
  with the following semantics:
  - {WaitTwo X Y} can return 1 if X is bound
  - {WaitTwo X Y} can return 2 if Y is bound
  - If either X or Y is bound, {WaitTwo X Y} will return
- If both X and Y are unbound, it just waits
- If both X and Y are bound, it can return either 1 or 2, both are possible (nondeterminism!)

14

Client/Server with WaitTwo

- Here is the client/server with WaitTwo:

```
proc {Server S1 S2}
     C={WaitTwo S1 S2}
in
     case C|S1|S2 of 1|(M1|T1)|S2 then
           (handle M1) {Server T1 S2}
     [] 2|S1|(M2|T2) then
           (handle M2) {Server S1 T2}
     end
end
```

- If Client 1 sends a message, C=1 and it is handled
- If Client 2 sends a message, C=2 and it is handled
- What happens if both Client 1 and Client 2 send messages?

15

WaitTwo is not scalable

- What happens if we have millions of clients?
  - WaitTwo solves the problem for two clients
  - How can we wait on millions of clients?
- One possibility is to "merge" all client streams into a single stream:

```
fun {Merge S1 S2}
     C={WaitTwo S1 S2}
in
     case C|S1|S2 of 1|(M1|T1)|S2 then M1|{Merge T1 S2}
     [] 2|S1|(M2|T2) then M2|{Merge S1 T2}
     end
end
```

- With Merge we build a huge tree of stream mergers. It must expand and contract if new clients arrive or old clients leave. Not very nice!

16

Solution 2: Ports

- A better solution is to add ports (named streams)
- Ports have two operations:
  P={NewPort S} % Create port P with stream S
  {Send P X}   % Add X to end of port P's stream
- How does this solve our problem?
  - With a million clients C_1 to $C_{1000000}$:
    Each client C_i does {Send M_i P} for each message it sends
  - The server reads the stream S, which contains all messages from all clients in some nondeterministic order

17

Port example operations

- We create a port and do sends:
  P={NewPort S}
  {Browse S} % Displays _
  {Send P a} % Displays a|_
  {Send P b} % Displays a|b|_
- What happens if we do:
  thread {Send P c} **end**
  thread {Send P d} **end**
- What are the possible results of these two sends for all choices of the scheduler?

18

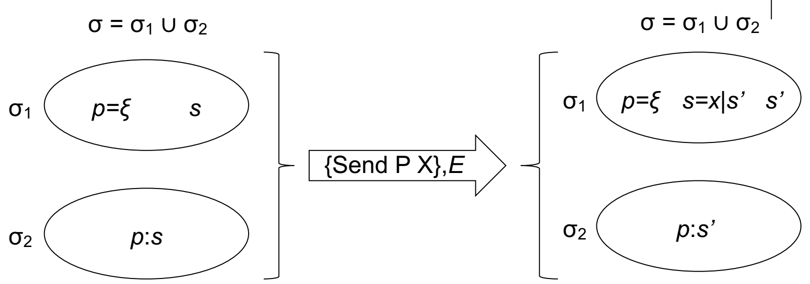
Port semantics (1)

- Assume single-assignment store σ_1 with variables
- Assume a port store σ_3 that contains pairs of variables

environment

- P={NewPort S}, {P→*p*, S→*s*}
  - Assume unbound variables p, $s \in \sigma_1$
  - Create fresh name ξ, bind $p=\xi$, add pair $p{:}s$ to σ_2
- {Send P X}, {P→*p*, X→*x*}
  - Assume $p=\xi$, unbound variable $s \in \sigma_1$, $p{:}s \in \sigma_2$
  - Create fresh unbound variable s', bind $s=x|s'$, update pair to $p{:}s'$

Port semantics (2)



- {Send X P} adds x to the end of the port's stream and updates the new end of stream
  - The send operation is atomic, which means the scheduler is guaranteed never to stop in the middle, so it happens as if it is one indivisible step
- We assume that environment E={P→*p*,X→*x*}

Client/server with ports

- Assume port P={NewPort S}
- Client code: (any number of clients!)
  - Do {Send P M} to send message to server
- Server code:

```
proc {Server S}
    case S of M|T then
        (handle M) {Server T}
    end
end
```

21

Message-passing concurrency

22

Message-passing concurrency

- Message-passing concurrency is a new paradigm for concurrent programming
  - It consists of deterministic dataflow with ports
  - It is also called multi-agent actor programming
- We will show how to write concurrent programs in this new paradigm
  - We will define port objects and active objects
  - We show how to do message protocols
  - We show how to write large multi-agent programs

Stateless port objects (stateless agents)

Stateless port objects

- A stateless port object is a combination of a port, a thread, and a recursive list function
  - We also call it a stateless agent
- Each agent is defined in terms of how it replies to messages
- Each agent has its own thread, so there are no problems with concurrency
- Agents are a very useful concept!

25

A math agent

- First the computation procedure:

```
proc {Math M}
   case M
   of add(N M A) then A=N+M
   [] mul(N M A) then A=N*M
   …
   end
end
```

26

Making it a port object

- We add a port, a thread, and a recursive procedure:

```
MP={NewPort S}
proc {MathProcess Ms}
    case Ms of M|Mr then
        {Math M} {MathProcess Mr}
    end
end
thread {MathProcess S} end
```

27

Using ForAll

- We replace MathProcess by ForAll:

```
proc {ForAll Xs P}
    case Xs of nil then skip
    [] X|Xr then {P X} {ForAll Xr P}
    end
end
```

- Using ForAll, we get:

```
proc {MathProcess Ms} {ForAll Ms Math} end
```

28

Defining new port objects (1)

- A generic way to build stateless port objects:

```
fun {NewAgent0 Process}
     Port Stream
in
     Port={NewPort Stream}
     thread {ForAll Stream Process} end
     Port
end
```


Defining new port objects (2)

- A generic way to build stateless port objects:

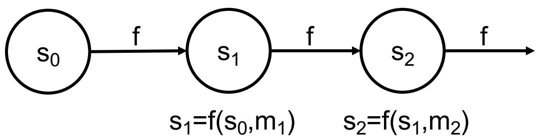
```
fun {NewAgent0 Process}
     Port Stream
in
     Port={NewPort Stream}
     thread for M in Stream do {Process M} end end
     Port
end
```

Using syntax of **for** loops

Stateful port objects (stateful agents)

31

Stateful port objects (Section 5.2)



- A stateful port object, also called stateful agent, has an internal memory s_i called its state
- The state is updated with each message received, which gives a state transition function:
  F: State × Msg ↦ State

32

Creating stateful port objects

- We define a generic function for stateful port objects:

```
fun {NewPortObject Init F}
    proc {Loop S State}
       case S of M|T then {Loop T {F State M}} end
    end
    P
in
    thread S in P={NewPort S} {Loop S Init} end
    P
end
```

33

Structure of Loop

- Does the Loop function ring a bell?

```
proc {Loop S State}
    case S of M|T then {Loop T {F State M}} end
end
```

- Loop starts from an initial state
- Loop successively applies F to the previous state and a message
- The function F is a binary operation
- …

34

Structure of Loop

- Does the Loop function ring a bell?

```
proc {Loop S State}
    case S of M|T then {Loop T {F State M}} end
end
```

- Loop starts from an initial state
- Loop successively applies F to the previous state and a message
- The function F is a binary operation
- Of course! It is a Fold operation!

35

FoldL operation

- FoldL is an important higher-order operation:

```
fun {FoldL S F U}
    case S
    of nil then U
    [] H|T then {FoldL T F {F U H}}
    end
end
```

36

Fold is the heart of the agent

- We replace:
  thread S **in** P={NewPort S} {Loop S Init} **end**
- by:
  thread S **in** P={NewPort S} {FoldL S F Init} **end**

- Oops! There is a small bug…

37

Updated NewPortObject

- We define a generic function for stateful port objects:

```
fun {NewPortObject Init F}
     P Out
in
     thread S in P={NewPort S} Out={FoldL S F Init} end
     P
end
```

- Out is the final state when the agent terminates
  - It never terminates here, but in another definition it might

38

Example Cell agent

- This agent behaves like a cell!

```
fun {CellProcess S M}
    case M
    of assign(New) then New
    [] access(Old) then Old=S S
    end
end
```

- Cells and ports are equivalent in expressiveness
  - Even though they look very different

39

Uniform interfaces (1)

- We can create and use a cell agent:

```
declare Cell
Cell={NewPortObject CellProcess 0}
{Send Cell assign(1)}
local X in {Send Cell access(X)} {Browse X} end
```

- We want to have the same interface as objects:

```
{Cell assign(1)}
local X in {Cell access(X)} {Browse X} end
```

40

Uniform interfaces (2)

- We change the output to be a procedure:

```
fun {NewPortObject Init F}
      P Out
in
      thread S in P={NewPort S} Out={FoldL S F Init} end
      proc {$ M} {Send P M} end
end
```

- P is hidden inside the procedure by lexical scoping
- This makes it easier to use port objects or standard objects as we saw before

41

Message protocols

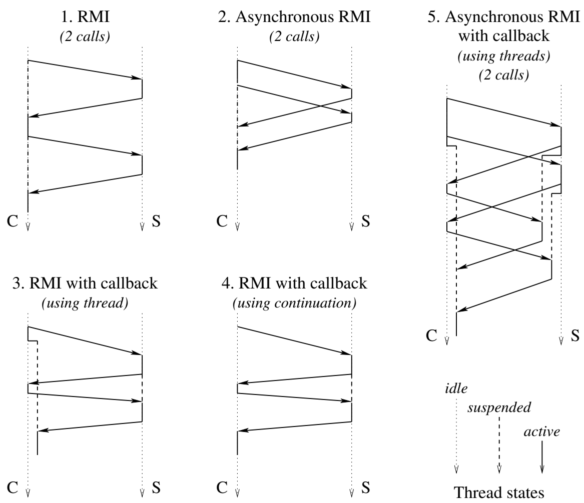
42

Message protocols (1)

- A message protocols is a sequence of messages between two or more parties that can be understood at a higher level of abstraction than individual messages
- Using port objects, let us investigate some important message protocols
- Code given during the lecture!
  - Also in Section 5.3 of the course book

43

Message protocols (2)



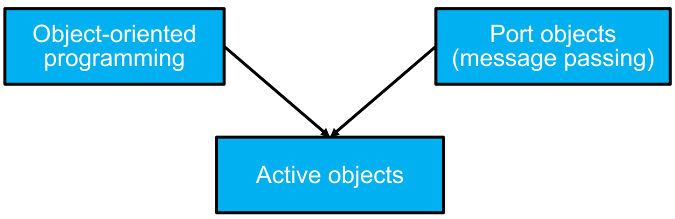
- We start with a simple RMI
- We then make it asynchronous and add callbacks
- The most complicated protocol is asynchronous RMI with callback

44

Active objects

45

Active objects (Section 7.8)



- An active object is a port object whose behavior is defined by a class
- Active objects combine the abilities of object-oriented programming (including polymorphism and inheritance) and message-passing concurrency
- To explain active objects, we refresh your memory on object-oriented programming and we introduce classes in Oz

46

Classes and objects in Oz

- We saw objects in the course
- We now complete this explanation by introducing classes and their Oz syntax

```
class Counter
      attr i
      meth init(X)
         i := X
      end
      meth inc(X)
         i := @i + X
      end
      meth get(X)
         X=@i
      end
end
```

- Create an object:

```
Ctr={New Counter init(0)}
```

- Call the object:

```
{Ctr inc(10)}
{Ctr inc(5)}
local X in
      {Ctr get(X)}
      {Browse X}
end
```

47

Defining active objects

- Active objects are defined by combining classes and port objects
- We use the uniform interface to make them look like standard Oz objects

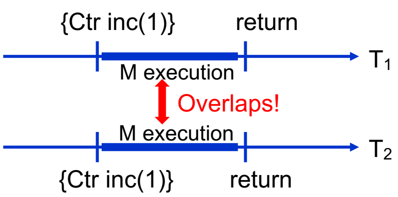
```
fun {NewActive Class Init}
      Obj={New Class Init}
      P
in
      thread S in
            {NewPort S P}
            for M in S do {Obj M} end
      end
      proc {$ M} {Send P M} end
end
```

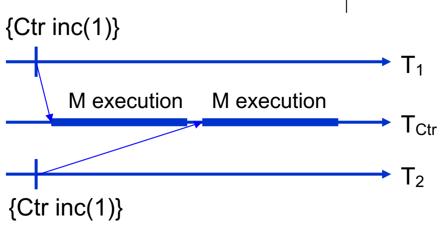
48

Passive objects and active objects

- We make an important distinction between passive objects and active objects
- Standard objects in Oz (and many other languages, such as Java and Python) are now called passive objects
  - This is because they execute in the thread of their caller; they do not have their own thread
- This is in contrast to active objects, which have their own thread
- Let us compare passive and active objects!

Concurrency comparison





- Passive objects cannot be safely called from more than one thread
- The method executions can overlap, which leads to concurrency bugs

- Active objects are completely safe when called from more than one thread
- The method executions are executed sequentially in the active object's own thread

Passive objects are not concurrency-safe!

- The following code is buggy:

```
Ctr={New Counter init(0)}
thread {Ctr inc(1)} end
thread {Ctr inc(1)} end
local X in
    {Ctr get(X)}
    {Browse X}
end
```

- This can display 1! Why?
  - Look at the instruction i := @i +1
  - If the scheduler puts T1 to sleep after @i and before i:=, executes T2 fully, and then resumes T1

- The following code is correct:

```
Ctr={NewActive Counter init(0)}
thread {Ctr inc(1)} end
thread {Ctr inc(1)} end
local X in
    {Ctr get(X)}
    {Browse X}
end
```

- This will always display 2
  - Because the two methods are executed sequentially by Ctr's thread

51

(to be continued)

52