CPL - Erlang

Marco Patrignani

KU Leuven

24 October 2014

Outline

Outline

What is Erlang?

- programming language +
- runtime system +
- OTP (libraries for DBs, FFIs ...)

What is Erlang?

- programming language +
- runtime system +
- OTP (libraries for DBs, FFIs ...)

Programming language:

What is Erlang?

- programming language +
- runtime system +
- OTP (libraries for DBs, FFIs ...)

Programming language:

- functional (like)

What is Erlang?

- programming language +
- runtime system +
- OTP (libraries for DBs, FFIs ...)

Programming language:

- functional (like)
- dynamically typed

(*unlike*)

What is Erlang?

- programming language +
- runtime system +
- OTP (libraries for DBs, FFIs ...)

Programming language:

- functional (like)
- dynamically typed

  (*unlike*)
- concurrent

What is Erlang?

- programming language +
- runtime system +
- OTP (libraries for DBs, FFIs ...)

Programming language:

- functional (like)
- dynamically typed

  (*unlike*)
- concurrent
- fault-tolerant

What is Erlang?

- programming language +
- runtime system +
- OTP (libraries for DBs, FFIs ...)

Programming language:

- functional (like)
- dynamically typed
  (*unlike*)
- concurrent
- fault-tolerant

Runtime system:

What is Erlang?

- programming language +
- runtime system +
- OTP (libraries for DBs, FFIs ...)

Programming language:

- functional (like)
- dynamically typed

  (*unlike*)
- concurrent
- fault-tolerant

Runtime system:

- garbage collector

  (like)

Why using Erlang?

- **inherently concurrent programs:**
    - achieved via processes (no threads)

Why using Erlang?

- **inherently concurrent programs:**
    - achieved via processes (no threads)
    - benefit from parallelisation (also thanks to SSA)

Why using Erlang?

- **inherently concurrent programs:**
  - achieved via processes (no threads)
  - benefit from parallelisation (also thanks to SSA)
- **distributed programs** on different machines

Why using Erlang?

- **inherently concurrent programs:**
  - achieved via processes (no threads)
  - benefit from parallelisation (also thanks to SSA)
- **distributed programs** on different machines
- **fault-tolerant systems:**
  several mechanisms to recover faults without a system crash

Why using Erlang?

- **inherently concurrent programs:**
  - achieved via processes (no threads)
  - benefit from parallelisation (also thanks to SSA)
- **distributed programs** on different machines
- **fault-tolerant systems:**
  several mechanisms to recover faults without a system crash
- **non-stop applications:**
  ability to load code at runtime

Why **NOT** using Erlang?

- **Poor support for frontends/GUIs**

Why **NOT** using Erlang?

- **Poor support for frontends/GUIs**
- **Not as supported as other languages** (unlike)

Why **NOT** using Erlang?

- **Poor support for frontends/GUIs**
- **Not as supported as other languages** (unlike)
- **Not known / understood as other languages** (unlike or C) (like)

Who uses Erlang?

Who uses Erlang?

facebook

ERICSSON

amazon.com®

WhatsApp

Erlang SOLUTIONS

YAHOO!®

MOTOROLA

Outline

Functional code

```
-module(app).

-export([func/1]).

func( Num )->
    Local_Var = 2 * Num,
    Avg = average([Num, Local_Var]),
    Sqr_Avg = math:sqrt(Avg),
    io:format("Result␣~p.␣~n",[Sqr_Avg]),
    ok.

average( L ) ->
    lists:foldr(fun(El, Acc)-> El + Acc end , 0, L).
```

Functional code

```
-module(app).  module definition

-export([func/1]).

func( Num )->
    Local_Var = 2 * Num,
    Avg = average([Num, Local_Var]),
    Sqr_Avg = math:sqrt(Avg),
    io:format("Result␣~p.␣~n",[Sqr_Avg]),
    ok.

average( L ) ->
    lists:foldr(fun(El, Acc)-> El + Acc end , 0, L).
```

Functional code

EVERYTHING ends with a dot "."

```
-module(app).

-export([func/1]).

func( Num )->
    Local_Var = 2 * Num,
    Avg = average([Num, Local_Var]),
    Sqr_Avg = math:sqrt(Avg),
    io:format("Result␣~p.␣~n",[Sqr_Avg]),
    ok.

average( L ) ->
    lists:foldr(fun(El, Acc)-> El + Acc end , 0, L).
```

Functional code

commas "," separate things

```
-module(app).

-export([func/1]).

func( Num )->
    Local_Var = 2 * Num ,
    Avg = average([Num , Local_Var]) ,
    Sqr_Avg = math:sqrt(Avg) ,
    io:format("Result␣~p.␣~n",[Sqr_Avg]),
    ok.

average( L ) ->
    lists:foldr(fun(El, Acc)-> El + Acc end , 0, L).
```

Functional code

```
-module(app).

-export([func/1]).        list of exported functions

func( Num )->
    Local_Var = 2 * Num,
    Avg = average([Num, Local_Var]),
    Sqr_Avg = math:sqrt(Avg),
    io:format("Result␣~p.␣~n",[Sqr_Avg]),
    ok.

average( L ) ->
    lists:foldr(fun(El, Acc)-> El + Acc end , 0, L).
```

Functional code

```
-module(app).

-export( [ func/1 ] ).   list of exported functions
                   lists are between square brackets “[ ] ”
func( Num )->
    Local_Var = 2 * Num,
    Avg = average([Num, Local_Var]),
    Sqr_Avg = math:sqrt(Avg),
    io:format("Result␣~p.␣~n",[Sqr_Avg]),
    ok.

average( L ) ->
    lists:foldr(fun(El, Acc)-> El + Acc end , 0, L).
```

Functional code

```
-module(app).

-export([func/1]).

func( Num )->
    Local_Var = 2 * Num,
    Avg = average([Num, Local_Var]),
    Sqr_Avg = math:sqrt(Avg),
    io:format("Result␣~p.␣~n",[Sqr_Avg]),
    ok.

average( L ) ->
    lists:foldr(fun(El, Acc)-> El + Acc end , 0, L).
```

list of exported functions

arity of the function

Functional code

```
-module(app).

-export([func/1]).

func( Num )->      function definition
    Local_Var = 2 * Num,
    Avg = average([Num, Local_Var]),
    Sqr_Avg = math:sqrt(Avg),
    io:format("Result␣~p.␣~n",[Sqr_Avg]),
    ok.

average( L ) ->
    lists:foldr(fun(El, Acc)-> El + Acc end , 0, L).
```

Functional code

```
-module(app).

-export([func/1]).

func( Num )->       function definition
    Local_Var = 2 * Num,
    Avg = average([Num, Local_Var]),
    Sqr_Avg = math:sqrt(Avg),
    io:format("Result␣~p.␣~n",[Sqr_Avg]),
    ok.

average( L ) ->   LOCAL function definition
    lists:foldr(fun(El, Acc)-> El + Acc end , 0, L).
```

Functional code

```
-module(app).

-export([func/1]).

func( Num )->
    Local_Var = 2 * Num,
    Avg = average([Num, Local_Var]),
    Sqr_Avg = math:sqrt(Avg),
    io:format("Result␣~p.␣~n",[Sqr_Avg]),
    ok.

average( L ) ->
    lists:foldr(fun(El, Acc)-> El + Acc end , 0, L).
```

variable, single assignment, like

Functional code

```
-module(app).

-export([func/1]).

func( Num )->
    Local_Var = 2 * Num, variable, single assignment, like
    Avg = average([Num, Local_Var]),
    Sqr_Avg = math:sqrt(Avg),
    io:format("Result␣~p.␣~n",[Sqr_Avg]),
    ok . atom

average( L ) ->
    lists:foldr(fun(El, Acc)-> El + Acc end , 0, L).
```

Functional code

```
-module(app).

-export([func/1]).

func( Num )->
    Local_Var = 2 * Num,
    Avg = average([Num, Local_Var]),
    Sqr_Avg = math:sqrt(Avg),
    io:format("Result␣~p.␣~n",[Sqr_Avg]),
    ok .

average( L ) ->
    lists:foldr(fun(El, Acc)-> El + Acc end , 0, L).
```

variable, single assignment, like

atom

mind the difference!!

Functional code

```
-module(app).

-export([func/1]).

func( Num )->
    Local_Var = 2 * Num,
    Avg = average([Num, Local_Var]), local function call
    Sqr_Avg = math:sqrt(Avg),
    io:format("Result␣~p.␣~n",[Sqr_Avg]),
    ok.

average( L ) ->
    lists:foldr(fun(El, Acc)-> El + Acc end , 0, L).
```

Functional code

```
-module(app).

-export([func/1]).

func( Num )->
    Local_Var = 2 * Num,
    Avg = average([Num, Local_Var]),
    Sqr_Avg = math:sqrt(Avg),
    io:format("Result␣~p.␣~n",[Sqr_Avg]),
    ok.

average( L ) ->
    lists:foldr(fun(El, Acc)-> El + Acc end , 0, L).
```

local function call

function call across modules

Functional code

```
-module(app).

-export([func/1]).

func( Num )->
    Local_Var = 2 * Num,
    Avg = average ([Num, Local_Var]),
    Sqr_Avg = math : sqrt (Avg),
    io:format("Result␣~p.␣~n",[Sqr_Avg]),
    ok.

average( L ) ->
    lists:foldr(fun(El, Acc)-> El + Acc end , 0, L).
```

local function call

function call across modules

Module name – colon “:” – function name

Functional code

```
-module(app).

-export([func/1]).

func( Num )->
    Local_Var = 2 * Num,
    Avg = average([Num, Local_Var]),
    Sqr_Avg = math:sqrt(Avg),
    io:format("Result␣~p.␣~n",[Sqr_Avg]),
    ok.

average( L ) ->
    lists:foldr(fun(El, Acc)-> El + Acc end , 0, L).
```

supports higher-order functions

(like , ,)

Functional code

```erlang
-module(app).

-export([func/1]).

func( Num )->
    Local_Var = 2 * Num,
    Avg = average([Num, Local_Var]),
    Sqr_Avg = math:sqrt(Avg),
    io:format("Result␣~p.␣~n",[Sqr_Avg]),
    ok.

average( L ) ->
    lists:foldr( fun(El, Acc)-> El + Acc end , 0, L).
```

anonymous function (like ,)

Erlang datatypes (selection of)

- Integers: 1, 15 and -4217

Erlang datatypes (selection of)

- Integers: 1, 15 and -4217
- Strings: "You", "are", "sleeping"

Erlang datatypes (selection of)

- Integers: 1, 15 and -4217
- Strings: "You", "are", "sleeping"
- Atoms: distinguished values

Erlang datatypes (selection of)

- Integers: 1, 15 and -4217
- Strings: "You", "are", "sleeping"
- Atoms: distinguished values
- Pids: process identifier

Erlang datatypes (selection of)

- Integers: 1, 15 and -4217
- Strings: "You", "are", "sleeping"
- Atoms: distinguished values
- Pids: process identifier
- Funs: function closures created by expressions: `fun(...) ->... end.`

Erlang datatypes (selection of)

- Integers: 1, 15 and -4217
- Strings: "You", "are", "sleeping"
- Atoms: distinguished values
- Pids: process identifier
- Funs: function closures created by expressions: `fun(...) ->... end.`
- Tuples: contain a fixed number data types: `{E1, E2, ..., En}`

Erlang datatypes (selection of)

- Integers: 1, 15 and -4217
- Strings: "You", "are", "sleeping"
- Atoms: distinguished values
- Pids: process identifier
- Funs: function closures created by expressions:
  `fun(...) ->... end.`
- Tuples: contain a fixed number data types:
  `{E1, E2, ..., En}`
- Lists: `[ Head | Tail ]`. `[]` denotes an empty list.

Dynamic typing

- NO static typing (unlike)

Dynamic typing

- NO static typing (unlike)
- This is a valid erlang program (will **fail** at runtime)

```
add( X, Y ) ->
    X + Y
end.
...
add( 5, "marco" ).
```

Compiling and running Erlang code

- Download and install the runtime:
  `http://www.erlang.org/download.html`

Compiling and running Erlang code

- Download and install the runtime: `http://www.erlang.org/download.html`
- `erl` starts the console

Compiling and running Erlang code

- Download and install the runtime: `http://www.erlang.org/download.html`
- `erl` starts the console
- `erlc filename.erl` compiles

Compiling and running Erlang code

- Download and install the runtime: `http://www.erlang.org/download.html`
- `erl` starts the console
- `erlc filename.erl` compiles
- run commands within the console

Compiling and running Erlang code

- Download and install the runtime: `http://www.erlang.org/download.html`
- `erl` starts the console
- `erlc filename.erl` compiles

- run commands within the console
- `c(filename).` compiles from the console

Useful links

- Erlang API: http://www.erlang.org/doc/man_index.html
- Part I of *Concurrent programming in Erlang*, by J. Armstrong, R. Virding, C. Wikström and M. Williams: http://www.erlang.org/download/erlang-book-part1.pdf
- http://www.erlang.org/static/getting_started_quickly.html
- http://www.erlang.org/doc/getting_started/users_guide.html
- http://learnyousomeerlang.com/

Coding time

Coding time

`length/1`,
dynamic type error with length/1, atom_to_list BIF,
`filter/2`,
anonymous functions, guards in functions,
`tailFilter/2`

Outline

The Actor model

- everything is an actor

The Actor model

- everything is an actor
- messages are the means of communication (asynchronous)

The Actor model

- everything is an actor
- messages are the means of communication (asynchronous)
- actors have mailboxes where messages are queued

The Actor model

- everything is an actor
- messages are the means of communication (asynchronous)
- actors have mailboxes where messages are queued
- actors **`send`** and **`receive`** messages (only 2 primitives)

Actors vs Threads

Actors

- (generally) context switched by the runtime
- message passing (asynchronous)
- no race conditons: no locking
- can deadlock
- benefit from SSA

Actors vs Threads

Actors

- (generally) context switched by the runtime
- message passing (asynchronous)
- no race conditons: no locking
- can deadlock
- benefit from SSA

Threads

- (generally) context switched by the OS
- shared memory (sync/async)
- race conditions: needs locking
- can deadlock
- rely on mutable state

Erlang's concurrency model, informally

Actor model:

- actors are *lightweight processes*, not OS processes, not threads

Erlang's concurrency model, informally

Actor model:

- actors are *lightweight processes*, not OS processes, not threads
- 309 words of memory when spawned (very small!)

Erlang's concurrency model, informally

Actor model:

- actors are *lightweight processes*, not OS processes, not threads
- 309 words of memory when spawned (very small!)
- fixed point context switches (optimal concurrency)

Erlang's concurrency model, informally

Actor model:

- actors are *lightweight processes*, not OS processes, not threads
- 309 words of memory when spawned (very small!)
- fixed point context switches (optimal concurrency)
- processes have mailboxes

Erlang's concurrency model, informally

Actor model:

- actors are *lightweight processes*, not OS processes, not threads
- 309 words of memory when spawned (very small!)
- fixed point context switches (optimal concurrency)
- processes have mailboxes
- send and **`receive`** are part of syntax.

has actors as a library

Akka implements actors for the JVM (, ...)
actor framework or Retlang for .Net

Receiving messages in Erlang

```
receive
    mess ->
        ok;
    {tuple, Number} ->
        io:format("Received␣~p",[Number])
    after
        10000 ->
            ok
end.
```

Receiving messages in Erlang

```
receive
    mess ->
        ok;
    {tuple, Number} ->
        io:format("Received␣~p",[Number])
    after
        10000 ->
            ok
end.
```

pattern matching (like ,)

Receiving messages in Erlang

```
receive
    mess ->
        ok ;
    {tuple, Number} ->
        io:format("Received␣~p",[Number])
    after
        10000 ->
            ok
end.
```

pattern matching (like ,)

separate by ";"

Receiving messages in Erlang

pattern matching (like ,)

last clause ends with NOTHING

```
receive
    mess ->
        ok;
    {tuple, Number} ->
        io:format("Received␣~p",[Number])
    after
        10000 ->
            ok
end.
```

Receiving messages in Erlang

timeout clause

```
receive
    mess ->
        ok;
    {tuple, Number} ->
        io:format("Received␣~p",[Number])
    after
        10000 ->
            ok
end.
```

The receive loop

- a receive loop is a recursive **`receive`** inside a function
- the recursive call must be *tail-recursive*

The receive loop

- a receive loop is a recursive **receive** inside a function
- the recursive call must be *tail-recursive*

```
server( NumMess )->
    receive
        mess ->
            server( NumMess + 1 );
        {tuple, Number} ->
            server( Number ) + 1
        after
            10000 ->
                ok
    end.
```

The receive loop

- a receive loop is a recursive **receive** inside a function
- the recursive call must be *tail-recursive*

```
server( NumMess )->
    receive
        mess ->
            server( NumMess + 1 ) ;
        {tuple, Number} ->
            server( Number ) + 1
        after
            10000 ->
                ok
    end.
```

this one is ok

The receive loop

- a receive loop is a recursive **receive** inside a function
- the recursive call must be *tail-recursive*

```
server( NumMess )->
    receive
        mess ->
            server( NumMess + 1 );
        {tuple, Number} ->
            server( Number ) + 1
        after
            10000 ->
                ok
    end.
```

this one is NOT

The receive loop

- a receive loop is a recursive **receive** inside a function
- the recursive call must be *tail-recursive*

State of the function (unlike , C)

```
server( NumMess )->
    receive
        mess ->
            server( NumMess + 1 );
        {tuple, Number} ->
            server( Number ) + 1
        after
            10000 ->
                ok
    end.
```

Sending messages in Erlang

- messages are sent to PIDs or NAMEs:

```
PID !{mess, Var, 1}
```

Sending messages in Erlang

- messages are sent to PIDs or NAMEs:

```
PID !{mess, Var, 1}
```

- PIDs are process identifiers

Sending messages in Erlang

- messages are sent to PIDs or NAMEs:

```
PID !{mess, Var, 1}
```

- PIDs are process identifiers
- processes are created with the BIF **spawn**/1-4, which returns a PID

Sending messages in Erlang

- messages are sent to PIDs or NAMEs:

```
PID !{mess, Var, 1}
```

- PIDs are process identifiers
- processes are created with the BIF **`spawn`**`/1-4`, which returns a PID
- the BIF **`self`**`/0` returns the PID of the current process

Sending messages in Erlang

- messages are sent to PIDs or NAMEs:

```
PID !{mess, Var, 1}
```

- PIDs are process identifiers
- processes are created with the BIF **spawn**/1-4, which returns a PID
- the BIF **self**/0 returns the PID of the current process
- PIDs can be registered to local names with **register**/2 (good for servers and for failing-respawning processes)

Sending messages in Erlang

- messages are sent to PIDs or NAMEs:

```
PID !{mess, Var, 1}
```

- PIDs are process identifiers
- processes are created with the BIF **`spawn`**`/1-4`, which returns a PID
- the BIF **`self`**`/0` returns the PID of the current process
- PIDs can be registered to local names with **`register`**`/2` (good for servers and for failing-respawning processes)
- local names are fetch with **`registered`**`/0`

Coding time

Coding time

receive messages, send messages, timeout

`spawn/3, register/2, unregister/1`

receive loop

Outline

The “Let it crash” philosophy

- expect failure
- deal with it

The "Let it crash" philosophy

- expect failure
- deal with it

Failures:

- in the same function: *exceptions, errors and exits.*

  Handled with `try / catch`, like in and

The "Let it crash" philosophy

- expect failure
- deal with it

Failures:

- in the same function: *exceptions, errors and exits*.

  Handled with `try` / **`catch`**, like in and
- in another process
  Handled as messages by *monitor/2* and *link/1*

Exceptions – throwing

Three types:

```
throw(Exception).
```

Exceptions – throwing

Three types:

```
throw(Exception).
erlang:error(Reason).
```

Exceptions – throwing

Three types:

```
throw(Exception).
erlang:error(Reason).
exit(Reason).
```

Exceptions – catching

```
throws(F) ->
    try F() of
        _ -> ok
    catch
        Throw -> {throw, caught, Throw};
        error:Error -> {error, caught, Error};
        exit:Exit -> {exit, caught, Exit}
    end.
```

Monitoring

- *Unidirectional*
- receive a message when a process dies:
  `{ 'DOWN', MonitorReference, process, Pid, Reason}`

Monitoring

- *Unidirectional*
- receive a message when a process dies:
  `{ 'DOWN', MonitorReference, process, Pid, Reason}`

```
Pid = spawn( function ),
Ref = monitor(process, Pid).
 ... % or
{ Pid , Ref } = spawn_monitor( function ).
... %remove with
demonitor( Ref ).
```

Linking

- *Bidirectional*
- receive message when *either* process dies:
  `{ 'EXIT', Pid, Reason}`
- only active after `process_flag(trap_exit, true)`

Linking

- *Bidirectional*
- receive message when *either* process dies:
  `{ 'EXIT', Pid, Reason}`
- only active after `process_flag(trap_exit, true)`

```
Pid = spawn( function ).
link( Pid ).
 ... % or
Pid = spawn_link( function ).
... %remove with
unlink( Pid ).
```

Coding time

Coding time

try/catch

`spawn_link/3, link/1, monitor/2`

Outline

What is distribution?

- processes on different Erlang nodes

What is distribution?

- processes on different Erlang nodes
- different Erlang nodes on different machines

What is distribution?

- processes on different Erlang nodes
- different Erlang nodes on different machines
- some applications are *inherently* distributed
  e.g. Cloud management, load balancing middleware …

Erlang Nodes

- an Erlang node is an executing Erlang system

Erlang Nodes

- an Erlang node is an executing Erlang system
- a node is given a name `erl -name asd`

Erlang Nodes

- an Erlang node is an executing Erlang system
- a node is given a name `erl -name asd`
- the BIF **`node`**`/0` returns the full name

Erlang Nodes

- an Erlang node is an executing Erlang system
- a node is given a name `erl -name asd`
- the BIF **`node`**`/0` returns the full name
- the BIFs **`spawn`**`/1-4`, `monitor/2`, **`link`**`/1`, **`register`**`/2` all work also with node names

Nodes

- connect with `net_kernel:connect_node( NodeName )`

Nodes

- connect with `net_kernel:connect_node( NodeName )`
- `net_kernel` coordinates distributed Erlang systems

Nodes

- connect with `net_kernel:connect_node( NodeName )`
- `net_kernel` coordinates distributed Erlang systems
- use cookies to prevent communications

Nodes

- connect with `net_kernel:connect_node( NodeName )`
- `net_kernel` coordinates distributed Erlang systems
- use cookies to prevent communications
- use `-hidden` to prevent communications

Coding time

Coding time

distributed communication

Conclusion

Erlang is:

- concurrent (also parallel and distributed)
- functional
- fail-resistant

Conclusion

Erlang is:

- concurrent (also parallel and distributed)
- functional
- fail-resistant
- good for backend software

Conclusion

Erlang is:

- concurrent (also parallel and distributed)
- functional
- fail-resistant

- good for backend software
- good for long-lived applications

Homework and lab exercises

Find the homework exercise in Toledo.
Lab sessions:

- November 4, from 1:30 PM to 4:00 PM, Location: 200A.SOL_Z
- November 14, from 10:30 AM to 1:00 PM, Location: 200A.SOL_Z

DO
your homework before the lab sessions