Introduction to Erlang

DS 2018 TP1

Eugenia Oshurko
Based on “Learn you some Erlang for great good!”
https://learnyousomeerlang.com/
Basic facts about Erlang

- Originally a proprietary language within Ericsson, later released as open source
- Was used for engineering telephone switches
- Functional
- Uses the actor model
- Used for developing systems that are
  - Distributed
  - Fault-tolerant
  - Soft real-time, highly available, non-stop applications
  - Allows hot swapping: code can be changed without stopping a system
- Not only a language, but an entire development environment. The code is compiled to bytecode and runs inside a virtual machine (VM).
Actor model

- Each actor is a separate process in the VM.
- No shared memory.
- Actors communicate strictly by sending messages.
- Every communication is explicit, traceable and safe.
- Erlang has three basic concurrency primitives (that we need to know about):
  - spawning new processes
  - sending messages
  - receiving messages
- A process has an identifier (PID), it is used as an address to communicate with the process.
- Processes do not return anything.
Concurrency

In the context of Erlang concurrency is many actors running independently, but not necessarily all at the same time.

Scaling

- Erlang is said to be able to scale in a directly proportional manner to how many cores your computer has
- This is usually not true
- It is possible, but most problems do not behave in a way that lets you just run everything at the same time
- Problems for which it is the case are called embarrassingly parallel
Erlang shell

- You can test most your stuff in Erlang shell; it will run your scripts when compiled and deployed.

```bash
$ erl
Erlang/OTP 20 [erts-9.2] [source] [64-bit] [smp:4:4] [ds:4:4:10]
[async-threads:10] [kernel-poll:false]

Eshell V9.2  (abort with ^G)
1> 1.
1
```

- Get help with `help()`.
- Exit `^C`. This won't help us if the shell freezes. Abort with `^G`.

User switch command

```bash
---> h
  c [nn]  - connect to job
  i [nn]  - interrupt job
  k [nn]  - kill job
  j        - list all jobs
  s [shell]  - start local shell
  r [node [shell]]  - start remote shell
  q        - quit erlang
  ? | h      - this message
-->
```
Erlang syntax 101

- Expressions have to be terminated with a period.
- Variables are capitalized.
- Variables are immutable.
Data-types

Standard:

- Numbers
- Atoms
- Tuples
- Lists

1> 2 + 15.
17
2> 5 / 2.
2.5
3> 5 div 2.
2
4> atom.
atom
5> atoms_rule.
atoms_rule
6> atoms_rule@erlang.
atoms_rule@erlang
7> 'Atoms can be cheated!'.
'Atoms can be cheated!'
8> atom = 'atom'.
atom
9> true and false.
false
10> not false.
true
11> X = 10, Y = 4.
4
12> Point = {X,Y}.
{10,4}
13> {U,V} = Point.
{10,4}
14> U.
10
15> [1, 2, 3, {numbers,[4,5,6]}, 5.34, atom].
[1,2,3,{numbers,[4,5,6]},5.34,atom]
16> [1,2,3] ++ [4,5].
[1,2,3,4,5]
17> [1,2,3] -- [1,2] -- [3].
[3]
18> hd([1,2,3,4]).
1
19> tl([[1,2,3,4]]).
[2,3,4]
20> List = [2,3,4].
[2,3,4]
21> NewList = [1|List].
[1,2,3,4]
22> [Head|Tail] = NewList.
[1,2,3,4]
23> Head.
1
24> Tail.
[2,3,4]
List comprehension

1> [2*N || N <- [1,2,3,4]].
   [2,4,6,8]
2> [X || X <- [1,2,3,4,5,6,7,8,9,10], X rem 2 =:= 0].
   [2,4,6,8,10]
3> RestaurantMenu = [{steak, 5.99}, {beer, 3.99}, {poutine, 3.50}, {kitten, 20.99}, {water, 0.00}].
   [{steak,5.99},
    {beer,3.99},
    {poutine,3.5},
    {kitten,20.99},
    {water,0.0}]
4> [{Item, Price*1.07} || {Item, Price} <- RestaurantMenu, Price >= 3, Price =< 10].
   [{steak,6.409300000000001},{beer,4.2693},{poutine,3.745}]
5> [X+Y || X <- [1,2], Y <- [2,3]].
   [3,4,4,5]
Other data-types

- Binaries
- Process identifiers

Various abstractions over lists and tuples:

- Strings
- Records
- Key-value stores (dictionaries)
- Arrays
- Directed graphs
- Queues
- Sets
- Ordered sets
- Sets of sets
- ....
Type system

- Erlang is **dynamically typed**
- Every error is caught at **runtime**
- Erlang is **strongly typed**: no implicit type conversions between terms.
- Erlang provides functions for explicit type conversion

```erlang
1> 6 + "1".
** exception error: bad argument in an arithmetic expression
  in operator +/2
  called as 6 + "1"
2> erlang:list_to_integer("54").
  54
3> erlang:integer_to_list(54).
  "54"
4> erlang:list_to_float("54.32").
** exception error: bad argument
  in function list_to_integer/1
  called as list_to_integer("54.32")
5> erlang:atom_to_list(true).
  "true"
```
Modules

- Modules are a collection of functions in a .erl single file.
- All functions in Erlang must be defined in modules.
- A function defined in a module is called as Module:Function(Arguments).

Shell

```
1> erlang:element(2, {a,b,c}).
b
2> element(2, {a,b,c}).
b
3> lists:seq(1,4).
[1,2,3,4]
4> seq(1,4).
** exception error: undefined
shell command seq/2
5> cd("path/to/").
"Path Name"
ok
6> c(mymodule).
{ok, mymodule}
7> mymodule:hello().
Hello, world!
ok
```

```
/path/to/mymodule.erl
-module(mymodule).
-export([hello/0,add/2]).

hello() ->
    io:format("Hello, world!~n").

add(A,B) ->
    A + B.
```
Pattern matching + Guards

- `;` separates function clauses
- `,' separates instructions

```erlang
% greet(male, Name) ->
io:format("Hello, Mr. ~s!", [Name]);
greet(female, Name) ->
io:format("Hello, Mrs. ~s!", [Name]);
greet(_, Name) ->
io:format("Hello, ~s!", [Name]).

same(X,X) -> true;
same(_,_) -> false.

polite_add(A,B) ->
io:format("Hello, let me add your numbers", [ ]),
A + B.

old_enough(X) when X >= 16 -> true;
old_enough(_) -> false.

wrong_age(X) when X < 16; X > 104 ->true;
wrong_age(_) -> false.
```
If/Case

- Ifs act like guards, but outside of a function clause's head. In fact, the if clauses are called Guard Patterns.

```
heh_fine() ->
  if 1 =:= 1 -> works end,
  if 1 =:= 2; 1 =:= 1 -> works end,
  if 1 =:= 2, 1 =:= 1 -> fails end.
```

This function will actually always throw an exception!

```
** exception error: no true branch found when evaluating an if expression in function ...
```

```
oh_god(N) ->
  if N =:= 2 -> might_succeed;
      true -> always_does  %% this is Erlang's if's 'else!' end.
end.
```

- Using a case ... of expression you can have the complex pattern matching you can use with each argument, and you can have guards on top of it.

```
beach(Temperature) ->
  case Temperature of
    {celsius, N} when N >= 20, N =< 45 ->
      'davorable';
    {kelvin, N} when N >= 293, N =< 318 ->
      'scientifically favorable';
    {fahrenheit, N} when N >= 68, N =< 113 ->
      'favorable in the US';
    _ ->
      'avoid beach'
  end.
```
Recursion

- As in most of functional programming languages there are no loops in Erlang, they are implemented using recursions.

\[
\begin{align*}
\text{fac}(0) & \rightarrow 1; \\
\text{fac}(N) \text{ when } N > 0 & \rightarrow N\times\text{fac}(N-1).
\end{align*}
\]

- Remember: always try to use tail recursion.
- To have a function call being tail recursive, it needs to be 'alone'.

\[
\begin{align*}
\text{tail_fac}(N) & \rightarrow \text{tail_fac}(N,1). \\
\text{tail_fac}(0,\text{Acc}) & \rightarrow \text{Acc}; \\
\text{tail_fac}(N,\text{Acc}) \text{ when } N > 0 & \rightarrow \\
\text{tail_fac}(N-1,N\times\text{Acc}).
\end{align*}
\]
Higher order functions

- As in all functional languages you can take a function you defined and then pass it as a parameter to another function.

```erlang
one() -> 1.
two() -> 2.
add(X,Y) -> X() + Y().
```

- Erlang provides syntax for anonymous functions.

```erlang
Add = fun(A, B) -> A + B end.
<Fun<erl_eval.12.99386804>>
2> Add(1,2).
3
3> lists:map(fun(X) -> X * 2 end, [1,2,3,4]).
[2,4,6,8]
```
Exceptions: errors, exits, throws

- Run-time error examples:
  - **function_clause**: all the guard clauses of a function failed, or none of the function clauses' patterns matched. Similar **case_clause/if_clause**.
  - **badmatch**: whenever pattern matching fails, e.g. you're trying to do impossible pattern matches, or just anything that isn't equal on both sides of the = operator.
  - **badarg**: calling functions with incorrect arguments.
  - **undef**: when you call a function that doesn't exist.

- Can raise custom run-time errors, e.g. `erlang:erlang:error(custom_error)`.

- Exits are triggered by an exit function being called in a process, they make the process stop its execution.

- Throws are used for cases that the programmer can be expected to handle.

### Code Examples

```erlang
sword(1) -> throw(slice);
sword(2) -> erlang:erlang:error(cut_arm);
sword(3) -> exit(cut_leg);
sword(4) -> throw(punch);
sword(5) -> exit(cross_bridge).

black_knight(Attack) when is_function(Attack, 0) ->
    try Attack() of
      _ -> "None shall pass."
    catch
      throw:slice -> "It is but a scratch."
      error:cut_arm -> "I've had worse."
      exit:cut_leg -> "Come on you pansy!"
      _:_ -> "Just a flesh wound."
    end.
```
Concurrency primitives: spawning processes

- A process runs a function and once it's done, it disappears.
- A process also has some hidden state (such as a mailbox for messages).
- Processes do not return anything.
- Every process has a unique id, called PID.
- Processes are concurrent
- Erlang VM can use multiple CPUs on multiprocessor machines

```erl
1> F = fun() -> 2 + 2 end. #Fun<erl_eval.20.67289768>
2> spawn(F).
<0.44.0>
3> G = fun(X) -> timer:sleep(10), io:format("~p~n", [X]) end. #Fun<erl_eval.6.13229925>
4> [spawn(fun() -> G(X) end) || X <- lists:seq(1,10)].
[<0.122.0>,<0.123.0>,<0.124.0>,<0.125.0>,<0.126.0>,<0.127.0>,<0.128.0>,<0.129.0>,<0.130.0>,<0.131.0>]
1
10
2
3
4
5
6
7
8
9
5> self().
<0.41.0>
6> exit(self()). ** exception exit: <0.41.0>
7> self().
<0.285.0>
```
Concurrency primitives: sending/receiving messages

- Sending message can be done using PID ! message ("!" is called bang symbol)
- Messages are sent asynchronously
- The sender continues immediately
- Any value can be sent as a message
- Each process has a message queue (mailbox) where incoming messages are placed.
- A process receives a message when it is extracted the mailbox.
- `receive` statement allows to receive messages.
  - Patterns and guards permit message selection
  - Receive-clauses are tried in order
  - If no message matches, the process suspends and waits for a new message

```
1> self() ! hello. hello
2> self() ! self() ! double. double
3> flush().
Shell got hello
Shell got double
Shell got double
ok

-module(kitchen).
-compile(export_all).

fridge1() ->
    receive
        {From, {store, _Food}} ->
            From ! {self(), ok},
            fridge1();
        {From, {take, _Food}} ->
            From ! {self(), not_found},
            fridge1();
        terminate -> ok
    end.
```
Concurrency primitives: useful facts

- Pids are often included in messages (\texttt{self}()), so that the receiver can reply to the sender.
- If the reply includes the Pid of the second process, it is easier for the first process to recognize the reply.
- A process terminates if
  - it finished the function call that it started with or
  - there was an exception that is not caught
- All messages sent to a terminated process are thrown away
- Same Pid will not be used for a long time
Registering processes

- A process can be registered under a name
- Any process can send a message to a registered process or look up the Pid
- The Pid might change (if the process is restarted and re-registered), but the name stays the same

```
1> Pid = spawn(mymodule, myfunction, []).  
2> register(mike, Pid).   
3> mike ! hello. 
```
Links and exit signals

- A **link** is a bidirectional relationship between any two processes. They are used to establish larger groups of processes that should all die together.
- When a process dies an exit **signal** `{ 'EXIT', B, Reason }` is sent to all linked processes, which are also killed.
- `{ 'EXIT', B, Reason }` exit cannot be caught with a try ... catch.
- Error propagation across processes is done through a process similar to message passing, but with a special type of message called signals.
- We can set `process_flag(trap_exit, true)` inside a process.
- When a process is trapping exits, it does not terminate when an exit signal is received. Instead, the signal is transformed into a message, which is just put into the mailbox.
- This way, a process can monitor other processes
Distributed Erlang

- A **distributed Erlang system** consists of a number of Erlang runtime systems communicating with each other.
- Each such runtime system is called a **node**.
- Message passing between processes at different nodes, as well as links and monitors, everything works the same through unique (across nodes) PIDs.
- Registered names are local to each node, can be still used to send messages e.g. `{mike, Node} ! Message.
- Nodes are connected the first time they try to communicate.
- The function `net_adm:ping(Node)` can be used to set up a connection between nodes.
  - returns `pong` (success) or `pang` (failure).
- We can start processes directly on another nodes.

```
$ erl -name node@eugenia-XPS13
Erlang/OTP 20 [erts-9.2] [source] [64-bit] [smp:4:4] [ds:4:4:10]
[async-threads:10] [kernel-poll:false]

Eshell V9.2  (abort with ^G)
(node@eugenia-XPS13)1> node().

'node@eugenia-XPS13'
```
Organisation & References

- 10 TPs + Project (graded)
- TPs can be found: [http://perso.ens-lyon.fr/ievgeniia.oshurko/teaching/DS/index.html](http://perso.ens-lyon.fr/ievgeniia.oshurko/teaching/DS/index.html)
- Project and its deadline will be announced
- Erlang docs [http://www.erlang.org/docs](http://www.erlang.org/docs)
- Book “Learn you some Erlang for great good!” [https://learnyousomeerlang.com/](https://learnyousomeerlang.com/) (free online)
Time to code...