

Advanced Programming

Introduction to Erlang

Ken Friis Larsen
`kflarsen@diku.dk`

Department of Computer Science
University of Copenhagen

October 3, 2017

Today's Buffet

- ▶ Erlang the language
- ▶ Kahoot!
- ▶ Concurrency-oriented programming
- ▶ (Distributed systems with Erlang)

These are slides that should have been lecture notes, but for now they are condemned to be neither.

Erlang — The Man

- ▶ Agner Krarup Erlang
- ▶ 1878–1929
- ▶ Invented **queueing theory** and **traffic engineering**, which is the basis for telecommunication network analysis.



Erlang Customer Declaration

Erlang is a:

- ▶ a concurrency-oriented language
- ▶ dynamically typed
- ▶ with a strict functional core language

Fundamental Stuff

- Integers and floating-points works as expected:

```
1> 21+21.
```

```
42
```

```
2> 3/4.
```

```
0.75
```

```
3> 5 div 2.
```

```
2
```

- We have lists:

```
4> [21,32,67] ++ [100,101,102].
```

```
[21,32,67,100,101,102]
```

- Strings are just lists of characters, and characters are just integers:

```
5> "Sur" ++ [112, 114, $i, $s, $e].
```

```
"Surprise"
```

Tuples and Atoms

- ▶ Erlang uses curly braces for tuples:

```
1> {"Bart", 9}.  
{"Bart",9}
```

- ▶ **Atoms** are used to represent non-numerical constant values (like enums in C and Java). Atom is a sequence of alphanumeric characters (including @ and _) that starts with a lower-case letter (or is enclosed in single-quotes):

```
2> bloody_sunday_1972.  
bloody_sunday_1972
```

```
3> [{bart@simpsons, "Bart", 9}, {'HOMER', "Homer", 42}].  
[{bart@simpsons,"Bart",9},{ 'HOMER', "Homer",42}]
```

Names and Patterns

- ▶ Names (variables) start with an upper-case letter.
- ▶ Like in Haskell and Prolog we use patterns to take things apart:

```
1> Homer = "Homer".
```

```
2> P = {point, 10, 42}.
```

```
3> [ C1, C2, C3 | Tail ] = Homer.
```

```
"Homer"
```

```
4> C2.
```

```
111
```

```
5> Tail.
```

```
"er"
```

```
6> {point, X, Y} = P.
```

```
{point,10,42}
```

```
7> X.
```

```
10
```

```
8> Y.
```

```
42
```

List Comprehensions

```
1> Digits = [0,1,2,3,4,5,6,7,8,9].  
[0,1,2,3,4,5,6,7,8,9]  
2> Evens = [ X || X <- Digits, X rem 2 == 0].  
[0,2,4,6,8]  
3> Cross = [{X,Y} || X <- [1,2,3,4], Y <- [11,22,33,44]].  
[{1,11}, {1,22}, {1,33}, {1,44},  
 {2,11}, {2,22}, ... ]  
4> EvenXs = [{X,Y} || {X,Y} <- Cross, X rem 2 == 0].  
[{2,11},{2,22},{2,33},{2,44},{4,11},{4,22},{4,33},{4,44}]
```


Maps

```
1> M = #{ name => "Ken", age => 43}.
#{age => 43,name => "Ken"}
2> ClunkyName = maps:get(name, M).
"Ken"
3> #{name := PatternName} = M.
4> PatternName.
"Ken"
5> #{name := Name, age := Age} = M.
#{age => 43,name => "Ken"}
6> {Name, Age}.
{"Ken", 43}
7> Wiser = M#{age := 44}.
#{age => 44,name => "Ken"}
8> WithPet = M#{pet => cat}.
#{age => 43,name => "Ken",pet => cat}
```

Functions

- Remember the move function from Exercise Set 0 (Haskell)?

```
move :: Direction -> Pos -> Pos
```

```
move North (x,y) = (x, y+1)
```

```
move West (x,y) = (x-1, y)
```

- In erlang:

```
move(north, {X, Y}) -> {X, Y+1};
```

```
move(west, {X, Y}) -> {X-1, Y}.
```

(note that we use semicolon to separate clauses, and period to terminate a declaration).

- Or naming a function literal:

```
Move = fun(Dir, {X,Y}) ->  
    case Dir of  
        north -> {X, Y+1};  
        west  -> {X-1, Y}  
    end  
end.
```

Modules

If we want to declare functions (rather than naming literals) then we need to put them in a **module**.

Modules are defined in `.erl` files, for example `erltest.erl`:

```
-module(erltest).
```

```
-export([move/2, qsort/1]).
```

```
move(north, {X, Y}) -> {X, Y+1};
```

```
move(west, {X, Y}) -> {X-1, Y}.
```

```
qsort([]) -> [];
```

```
qsort([Pivot|Rest]) ->
```

```
    qsort([X || X <- Rest, X < Pivot])
```

```
    ++ [Pivot] ++
```

```
    qsort([X || X <- Rest, X >= Pivot]).
```

Compiling Modules

- ▶ Using the function `c`, we can compile and load modules in the Erlang shell:

```
1> c(erltest).  
{ok,erltest}
```

- ▶ We can now call functions from our module:

```
2> erltest:qsort([101, 43, 1, 102, 24, 42]).  
[1,24,42,43,101,102]
```

- ▶ Or use them with functions from the standard library:

```
3> Moves = [{north, {1,1}}, {west, {43,42}},  
            {north, {23,22}}].  
4> lists:map(fun({Dir,Pos}) ->  
              erltest:move(Dir, Pos) end,  
            Moves).  
[{1,2},{42,42},{23,23}]
```

Define your favourite function

Make a module, `fib`, with your favourite (recursive) function.

Exceptions

- ▶ We can catch exceptions using `try`:

```
try Expr of  
  Pat1 -> Expr1;  
  Pat2 -> Expr2;  
  ...  
catch  
  ExPat1 -> ExExpr1;  
  ExPat2 -> ExExpr2;  
  ...  
after  
  AfterExpr  
end
```

- ▶ And we can throw an exception using `throw`:

```
throw(Expr)
```

Exceptional Moves

```
-module(exceptional_moves).  
-export([move/2,ignore_invalid/2]).  
  
move(north, {X, Y}) -> {X, Y+1};  
move(west, {0, _}) -> throw(invalid_move);  
move(west, {X, Y}) -> {X-1, Y}.  
  
ignore_invalid(Dir, Pos) ->  
    try move(Dir, Pos)  
    catch  
        invalid_move -> Pos  
    end.
```

Algebraic Data Types

- ▶ In Erlang we use tuples and atoms to build data structures.
- ▶ Representing trees in Haskell

```
data Tree a = Leaf | Node a (Tree a) (Tree a)  
t = Node 6 (Node 3 Leaf Leaf) (Node 9 Leaf Leaf)
```

- ▶ Representing trees in Erlang

```
T = {node, 6, {node, 3, leaf, leaf},  
      {node, 9, leaf, leaf}}.
```


Traversing Trees

- ▶ in Haskell:

```
contains _ Leaf = False  
contains key (Node k left right) =  
    if key == k then True  
    else if key < k then contains key left  
    else contains key right
```

- ▶ in Erlang:

```
contains(_, leaf) -> false;  
contains(Key, {node, K, Left, Right}) ->  
    if Key == K -> true;  
    Key < K    -> contains(Key, Left);  
    Key > K    -> contains(Key, Right)  
end.
```

Binary Data

- ▶ Erlang have outstanding support for working with raw byte-aligned data (**binaries**)
- ▶ $\langle\langle b_1, b_2, \dots, b_n \rangle\rangle$ is an n -byte value
 - ▶ 8-bit: $\langle\langle 111 \rangle\rangle$
 - ▶ 32-bit: $\langle\langle 0, 0, 0, 0 \rangle\rangle$
 - ▶ 40-bit: $\langle\langle \text{"Homer"} \rangle\rangle$
- ▶ Bit Syntax is used to pack and unpack binaries, here we can specify the size and encoding details (like endianness, for instance) for each element of the binary
 - ▶ General form:

$$\langle\langle E_1, E_2, \dots, E_n \rangle\rangle$$

where each element E have the form:

$$V : \text{size/type}$$

where V is a value and *size* and *type* can be omitted.

8-Bit Colour

- ▶ Suppose we need to work with 8-bit colour images, encoded in RGB format with 3 bits for the red and green components and 2 bits for the blue component.
- ▶ Pack and unpack functions:

```
pack8bit(R,G,B) ->  
  <<R:3,G:3,B:2>>.
```

```
unpack8bit(<<R:3,G:3,B:2>>) ->  
  {R, G, B}.
```

Concurrency-Oriented Programming

- ▶ The world is concurrent
- ▶ Thus, when we write programs that model or interact with the world concurrency should easily be modelled

Parallelism \neq Concurrency

- ▶ Parallelism

- ▶ use multiple CPUs to perform a computation
- ▶ maximise speed

- ▶ Concurrency

- ▶ Model and interact with the world
- ▶ minimise latency

Concurrency In Erlang

- ▶ Processes are lightweight and independent
- ▶ Processes can only communicate through message passing
- ▶ Message passing is fast
- ▶ Message passing is asynchronous (mailbox model)

Processes

- ▶ Processes can only communicate through message passing
- ▶ All processes have a unique process ID (pid)
- ▶ Any value can be send (serialization)
- ▶ We can `send` messages:

`Pid ! Message`

(we can get our own pid by using the build-in function `self/0`)

Receiving messages

- ▶ Mailbox ordered by arrival – *not* send time
- ▶ We can **receive** messages:

```
receive
  Pat1 -> Expr1;
  Pat2 when ... -> Expr2;
  ...
after
  Time -> TimeOutExpr
end
```

times-out after Time milliseconds if we haven't received a message matching one of Pat1, Pat2 with side condition,

Spawning Processes

- ▶ We can `spawn` new processes:

```
Pid = spawn(Fun)
```

or

```
Pid = spawn(Module, Fun, Args)
```

Concurrency Primitives, Summary

- ▶ We can **spawn** new processes:

```
Pid = spawn(Fun)
```

(we can get our own pid by using the build-in function `self`)

- ▶ We can **send** messages:

```
Pid ! Message
```

- ▶ We can **receive** messages:

```
receive
```

```
  Pat1 -> Expr1;
```

```
  Pat2 -> Expr2;
```

```
  ...
```

```
after
```

```
  Time -> TimeOutExpr
```

```
end
```

where we get a time-out after `Time` milliseconds if we haven't received a message matching one of `Pat1`, `Pat2`,

Let's make a concurrent version of the recursive function from `fib` (the one we made earlier).

Dealing With State

- ▶ Functions are pure (stateless).
- ▶ Processes are stateful.
- ▶ We organise our code as micro-servers that manage a state which can be manipulated via a client API.

Client-Server Basic Set Up

- ▶ We often want computations to be made in a server process rather than just in a function.
- ▶ That is, we start with something like the following template:

```
start() -> spawn(fun loop/0).  
request_reply(Pid, Request) ->  
  Pid ! {self(), Request},  
  receive  
    {Pid, Response} -> Response  
  end.  
loop() ->  
  receive  
    {From, Request} ->  
      From ! {self(), ComputeResult(Request)},  
      loop();  
    {From, Other} ->  
      From ! {self(), {error, Other}},  
      loop()  
  end.
```

Example: Move Server

```
start() -> spawn(fun loop/0).
```

```
move(Pid, Dir, Pos) -> request_reply(Pid, {move, Dir, Pos}).
```

```
request_reply(Pid, Request) ->  
  Pid ! {self(), Request},  
  receive  
    {Pid, Response} -> Response  
  end.
```

```
loop() ->  
  receive  
    {From, {move, north, {X, Y}}} ->  
      From ! {self(), {X, Y+1}},  
      loop();  
    {From, {move, west, {X, Y}}} ->  
      From ! {self(), {X-1, Y}},  
      loop()  
  end.
```

Student activity: Count Server

- ▶ Let's make a server that can keep track of a counter
- ▶ What is the client API?
- ▶ What is the internal state?

Example: Phone-Book, Interface

```
start() -> spawn(fun() -> loop(#{}) end).
```

```
add(Pid, Contact) ->  
  request_reply(Pid, {add, Contact}).
```

```
list_all(Pid) ->  
  request_reply(Pid, list_all).
```

```
update(Pid, Contact) ->  
  request_reply(Pid, {update, Contact}).
```


Example: Phone-Book, Implementation 1

```
loop(Contacts) ->
  receive
    {From, {add, Contact}} ->
      {Name, _, _} = Contact,
      case maps:is_key(Name, Contacts) of
        false ->
          From ! {self(), ok},
          loop(Contacts#{Name => Contact});
        true ->
          From ! {self(), {error, Name, is_already_there}},
          loop(Contacts)
      end;
  end;
```

Example: Phone-Book, Implementation 2

```
{From, list_all} ->
    List = maps:to_list(Contacts),
    From ! {self(),
            {ok, lists:map(fun({_, C}) -> C end, List)}},
    loop(Contacts);
{From, {update, Contact}} ->
    {Name,_,_} = Contact,
    NewContacts = maps:remove(Name, Contacts),
    From ! {self(), ok},
    loop(maps:put(Name, Contact, NewContacts));
{From, Other} ->
    From ! {self(), {error,unknown_request, Other}},
    loop(Contacts)
end.
```

Distributed Programming

- ▶ Simple definition: A distributed system is a system that involves at least two computers that communicate.
- ▶ Two models:
 - ▶ Closed world: Distributed Erlang, Java's RMI, .NET Remoting
 - ▶ Open world: IP Sockets
- ▶ Why distribute a system?
 - ▶ Inherently
 - ▶ Reliability
 - ▶ Scalability
 - ▶ Performance

Distributed Programs in Erlang

- ▶ *Distributed Erlang* for tightly coupled computers in a secure environment.
 - ▶ `spawn(Node, Fun)` to spawn a process running `Fun` on `Node`
 - ▶ `{RegAtom, Node} ! Mesg` sends `Mesg` to the process registered as `RegAtom` at `Node`.
 - ▶ `monitor_node(Node, Flag)` register the calling process for notification about `Node` if `Flag` is `true`; if `Flag` is `false` then monitoring is turned off.
- ▶ *Sockets* for untrusted environments:
 - ▶ To build a middle-ware layer for Erlang nodes
 - ▶ For inter-language communication.

See the documentation for `gen_tcp` and `gen_udp`

Setting Up Some Erlang Nodes

- ▶ To start nodes on the same machine, start `erl` with option `-sname`
- ▶ To start nodes on different machines, start `erl` with options `-name` and `-setcookie`:
 - ▶ On machine A:
`erl -name bart -setcookie BoomBoomShakeTheRoom`
 - ▶ On machine B:
`erl -name homer -setcookie BoomBoomShakeTheRoom`
- ▶ `rpc:call(Node, Mod, Fun, Args)` evaluates `Mod:Fun(Args)` on `Node`. (See the the manual page for `rpc` for more information.)

Common Erlang Pitfalls

- ▶ Variables starts with an upper-case letter, atoms starts with a lower-case letter.
- ▶ if expressions (you need to understand what a *guard expression* is).
- ▶ Misunderstanding of how patterns works.
- ▶ Functions starts processes, processes runs functions, functions are defined in modules.
- ▶ Not realising when to use asynchronous communication and when to use synchronous communication.

Summary

- ▶ Parallelism is not the same as concurrency.
- ▶ Share-nothing (that is, immutable data) and message passing takes a lot of the pain out of concurrent programming.
- ▶ Study `phonebook.erl` for a short tour of Erlang.
- ▶ Learning opportunity: See if you can make the code shorter or simpler by using `map` syntax rather than library functions.