Advanced Programming

Introduction to Erlang

Oleks Shturmov
oleks@oleks.info

Slides adapted from Ken Friis Larsen
kflarsen@diku.dk

Department of Computer Science
University of Copenhagen

September, 2021

Before we begin

- **Windows users**
  - Add `C:\Program Files\erl-24.0\bin` to your PATH environment variable
- **Everyone**

```
dialyzer --build_plt --apps erts kernel stdlib \
  crypto compiler
```

- Dialyzer is a tool for type-checking Erlang code
- This may take up to 20 minutes, exactly enough

Part I

Introduction

The Erlang Programming Language

- Developed in the early 1980s, by the guys above[1]
- All while working at Ericsson, programming telephone switches
- Useful for distributed, fault-tolerant systems, in general
- Open-sourced in 1998, still maintained by Ericsson
- Used today by WhatsApp (Facebook), Nintendo, Discord, etc.

[1] Image source: `https://www.youtube.com/watch?v=rYkI0_ixRDc`

Erlang Customer Declaration

Erlang is a:

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

On the matter of Erlang syntax

- Syntax heavily inspired by the Prolog programming language
- **Semantically, Erlang bears little resemblance to Prolog**
  - Prolog is a *logic programming language*
  - A distinct programming paradigm, beyond the scope of this course
- We won't delve on this history, except to justify peculiar syntax

Part II

Basic Concepts

`erl`

On Windows, the Erlang executable

- Starts an Erlang runtime system (i.e., an Erlang node)
  - Distributed (multi-node) Erlang is beyond the scope of this course
- Drops you into an Erlang shell for that node

```
$ erl
Erlang/OTP 24 [erts-12.0.3] [source] [64-bit] [...]

Eshell V12.0.3  (abort with ^G)
1> 21+21.
42
2>
```

- Use one of the following options to quit/kill `erl`:
  - Enter the command `q().`
  - Press `Ctrl+g`, and enter the command `q` (quit)
  - Press `Ctrl+c`, and enter the command `a` (abort)

Fundamental Stuff

- We have integers, floating-point numbers, and arithmetic:

```
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].
```

Fundamental Stuff

- We have integers, floating-point numbers, and arithmetic:

```
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"
```

Names (Variables)

- Names (variables) start with an upper-case letter

```
1> Homer = "Homer".
Homer
2> X=5, Y=2, X*Y.
10
```

- Once assigned, variables cannot be re-assigned[2]

```
3> X=3, Y=2, X*Y.
** exception error: no match of right hand side value 3
```

[2] Perhaps another relic of the Prolog past, or part of the "strict functional core"

What to do if you mess up in `erl`?

- `Eshell` is quite forgiving — you can just let exceptions happen
- In non-exceptional cases[3], you might be tempted to kill `erl`…
- A better option is to press `Ctrl+g`:

```
4> X/0
4> % Pressing Ctrl+g ...
User switch command
 -> h
  c [nn]            - connect to job
  i [nn]            - interrupt job
  ...
  q                 - quit erlang
  ? | h             - this message
 -> i
 -> c
** exception exit: killed
4> X/2.
2.5
```

- Bonus: you get to keep your names (variables)!

[3] Long-running, live-, or dead-locked commands

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}]
```

Patterns

- As in Haskell, we can use patterns to take things apart:

```
1> P = {point, 10, 42}.
2> [ C1, C2, C3 | Tail ] = "Homer".
"Homer"
3> C2.
111
4> Tail.
"er"
5> {point, X, Y} = P.
{point,10,42}
6> X.
10
7> 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 => 45}.
#{age => 45, name => "Ken"}
2> ClunkyName = maps:get(name, M).
"Ken"
3> #{name := PatternName} = M.
4> PatternName.
"Ken"
5> #{name := Name, age := Age} = M.
#{age => 45, name => "Ken"}
6> {Name, Age}.
{"Ken", 45}
7> Wiser = M#{age := 46}.
#{age => 46, name => "Ken"}
8> WithPet = M#{pet => {cat, "Toffee"}}.
#{age => 46, name => "Ken", pet => {cat, "Toffee"}}
```

Functions

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

```haskell
move :: Direction -> Pos -> Pos
move North (x,y) = (x, y+1)
move West  (x,y) = (x-1, y)
```

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).

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 `somemodule.erl`:

```
-module(warmup).
-export([move/2]).

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

- Note, how we **specify the arity** of functions on export

Compiling Modules

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

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

- We can now call functions from our module:

```
2> warmup:move(north, {0,0}).
{0,1}
```

- 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}) ->
                 warmup:move(Dir, Pos) end,
              Moves).
[{1,2},{42,42},{23,23}]
```

Part III

Type Checking

Erlang is dynamically typed

- The type of a value is determined at runtime
- Using values of wrong types induces runtime errors
- Perhaps, another relic of the Prolog past…
- However, static typing is principally complicated by Erlang's liberal communication primitives…

Enter Dialyzer and TypEr

- Dialyzer and TypEr offer a **gradual type system** for Erlang
- Part of an Erlang code optimization effort
- The more types you know, the more can you optimize

Enter Dialyzer and TypEr

- Dialyzer and TypEr offer a **gradual type system** for Erlang
- Part of an Erlang code optimization effort
- The more types you know, the more can you optimize

- Types form a (mathematical) lattice
- The top type is called `any()`
- All Erlang types conform to this type
- So we can safely give any value the type `any()`
- However, we can *gradually* increase the specificity of the types in our Erlang code as it evolves

any()

- One straight-forward, albeit not very useful way to type `move/2`:

```
-module(warmup).
-export([move/2]).

-spec move(any(), any()) -> any().
move(north, {X, Y}) -> {X, Y+1};
move(west, {X, Y}) -> {X-1, Y}.
```

any()

- One straight-forward, albeit not very useful way to type `move/2`:

```
-module(warmup).
-export([move/2]).

-spec move(any(), any()) -> any().
move(north, {X, Y}) -> {X, Y+1};
move(west, {X, Y}) -> {X-1, Y}.
```

- Let's see if Dialyzer would be happy:

```
$ erlc +debug_info warmup.erl
$ dialyzer warmup.beam
  Checking whether the PLT ... is up-to-date... yes
  Proceeding with analysis... done in 0m0.10s
done (passed successfully)
```

No underspecs, thanks

- Let's make Dialyzer a bit more pedantic

```
$ erlc +debug_info warmup.erl
$ dialyzer -Wunderspecs warmup.beam
  Checking whether the PLT ... is up-to-date... yes
  Proceeding with analysis...
warmup.erl:8:2: Type specification warmup:move
          (any(), any()) -> any() is a supertype of
            the success typing: warmup:move
          ('north' | 'west', {_, _}) -> {_, _}
 done in 0m0.14s
```

Let’s try to TypeEr

- TypEr is a Type-annotator for Erlang programs
- Let’s see how well it can type `move/2`:

```
$ typer warmup.erl
%% File: "warmup.erl"
%% ------------------
-spec move('north' | 'west' ,{_,_}) -> {_,_}.
```

- `'north' | 'west'` is definitely more specific than `any()`!
- `{_,_}` is a bit more specific than `any()`, but not terribly so
- Type inference in Erlang/OTP, in general, is rather weak
  - Using ‘+’ does not induce numeric types for `X` and `Y`

Singleton and Predefined Types

- A singleton type is an Erlang term
  - For instance, `true`, `false`, `1`, `42`, "Homer", `{}` (empty tuple), `[]` (empty list), `#{}` (empty map)
- Predefined types represent infinite sets of terms

Type	**Meaning**
`any()`	Any value
`atom()`	Any atom
`integer()`	Any integer[4]
`tuple()`	Any tuple
`float()`	Any float
`fun()`	Any function
`pid()`	Any process identifier

[4] Erlang uses arbitrary-precision arithmetic

Composite Types

- Composed using, among others, the following forms:

Syntax	Name	Example
T_1 '|' $\cdots$ '|' T_n	Union	`integer() | float()`
'[' T ']'	List	`[integer()]`
'{' T_1 ',' $\cdots$ ',' T_n '}'	Tuple	`{integer(), integer()}`
'#{' T_1 '=>' T_1' ',' $\cdots$ '}'	Map	`#{atom() => integer()}`

- Some predefined composite types:

Type	Defined as
`boolean()`	`true | false`
`list()`	`[any()]`
`string()`	`[char()]`
`map()`	`#{any() => any()}`

- For more, see Erlang/OTP documentation[5]

[5] http://erlang.org/documentation/doc-12.0/doc/reference_manual/typespec.html#builtin_types

Function Types

$$\text{`fun('}\ \text{`('}\ T_1^{param}\ \text{`,'}\ \ldots\ \text{`,'}\ T_m^{param}\ \text{`)'}\ \text{`->'}\ T^{res}\ \text{`)'}$$

- Consider the function literal

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

- Valid typings include:
  - `fun(any(), any()) -> any()`
  - `fun(atom(), tuple()) -> tuple()`
  - `fun(atom(), {integer(), integer()})`
    `-> {integer(), integer()}`

Typespecs

- A substantially better typing of `move/2`:

```
-module(warmup).
-export([move/2]).

-type position() :: {integer(), integer()}.
-type direction() :: north | west | east | south.

-spec move(direction(), position()) -> position().
move(north, {X, Y}) -> {X, Y+1};
move(west, {X, Y}) -> {X-1, Y}.
```

Part IV

Less Basic Concepts

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

```erlang
-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, or lack thereof

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

```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

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

Traversing Trees

- in Haskell:

```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:

```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)
- <<$b_1, b_2, \ldots, b_n$>> is an n-byte value
  - 8-bit: `<<111>>`
  - 32-bit: `<<0,0,0,0>>`
  - 40-bit: `<<"Homer">>`
- Bit Syntax is used to pack and unpack binaries, here we can specify the size and encoding details (like endianess, for instance) for each element of the binary
  - General form:

    <<$E_1, E_2, \ldots, E_n$>>

    where each element E have the form:

    $V : size/type$

    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.

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}.
```

Part V

Concurrency Primitives

Concurrency-Oriented Programming

- The world is concurrent
  - Many different things happen at the same time
- When we write programs that model, or interact with the world, concurrency should be easy to model

Parallelism $\neq$ Concurrency

- Parallelism
  - Bursts of non-interruptible computation,
    running on multiple processing units
  - Fixed synchronization points
  - Examples: GPUs, TPUs
  - Maximise amount of computation per clock cycle
- Concurrency
  - Interleaving threads of execution,
    running on one, or multiple processing units
  - Sporadic communication
  - Suitable for modeling, and interacting with the world
  - Minimise latency

Concurrency In Erlang

- Structure system in terms of lightweight, independent processes
- 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 sent and received (serialization)

Processes

- Processes can only communicate through message passing
- All processes have a unique process ID (pid)
- Any value can be sent and received (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

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, ...
- `receive ... end` is an expression (just like `case` and `if`).

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)
  ```

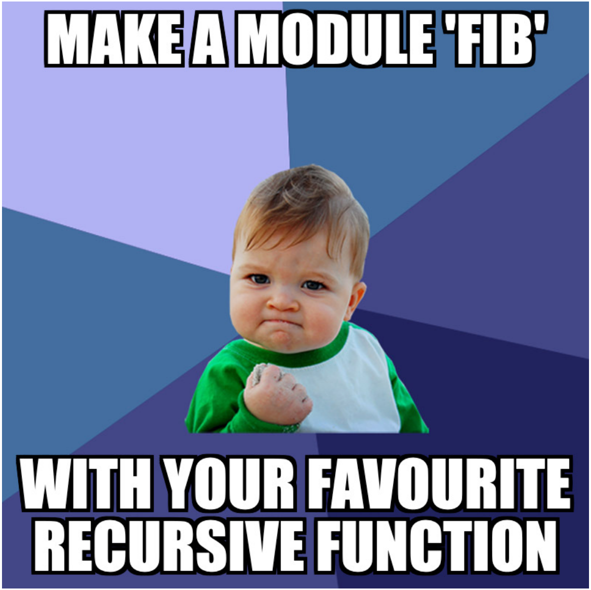
  (we can get our own pid by using the build-in function `self`)
- We can send messages:

  ```
  Pid ! Message
  ```
- We can receive messages:

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

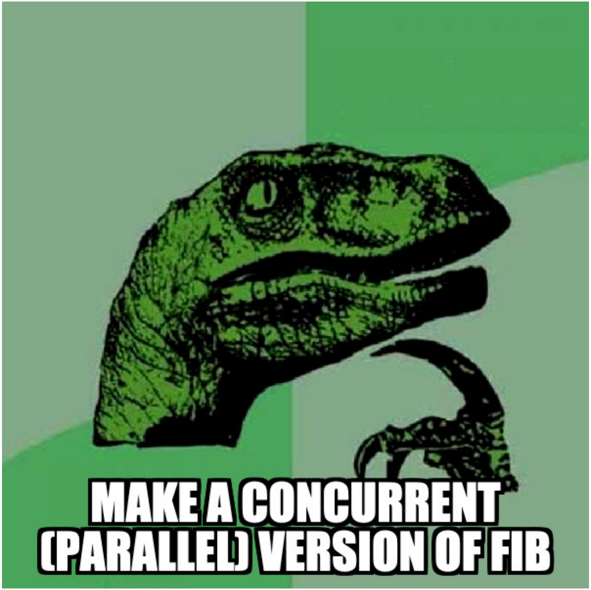
  where we get a time-out after `Time` milliseconds if we haven't received a message matching one of `Pat1`, `Pat2`, ...

Student Activation: Define your favourite function



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

Student Activation: Make It Concurrent



Part VI

State – How To Deal With It

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, OtherReq} ->
            From ! {self(), ComputeOther(OtherReq)},
            loop()
    end.
```

Example: Position Server

- Make a server that can keep track of a *position*.
- We should be able to:
  - `move` the position in some direction
  - `get_pos` to get the position

Example: Position Server, Implementation

```
start(Start) -> spawn(fun () -> loop(Start) end).
move(Pid, Dir) -> request_reply(Pid, {move, Dir}).
get_pos(Pid) -> request_reply(Pid, get_pos).

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

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

Student Activivation: 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: Todo-List, Interface

```erlang
start() -> spawn(fun() -> loop([]) end).

add_item(Pid, Description, Due) ->
    request_reply(Pid, {add, {Description, Due}}).

all_items(Pid) ->
    request_reply(Pid, all_items).

finish(Pid, Index) ->
    request_reply(Pid, {finish, Index}).
```

Example: Todo-List, Internal loop

```
loop(Items) ->
    receive
        {From, {add, {Description, Due}}} ->
            Item = #{ description => Description, due => Due},
            From ! {self(), ok},
            loop([Item | Items]);
        {From, all_items} ->
            From ! {self(), {ok, Items}},
            loop(Items);
        {From, {finish, Index}} ->
            Len = length(Items),
            if  Index =< 0; Len < Index ->
                    From ! {self(), {error, index_out_of_bounds}},
                    loop(Items);
                Index > 0, Index =< Len ->
                    {L1, [_ | L2]} = lists:split(Index-1, Items),
                    From ! {self(), ok},
                    loop(L1++L2)
            end
    end.
```

Part VII

Summary

Common Erlang Pitfalls

- Variables starts with an upper-case letter, atoms starts with a lower-case letter.
- Erlang does not have statements, only expressions.
- `if` expressions (you need to understand what a *guard expression* is).
- Misunderstanding 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
- Erlang code is hard to type check, so test it!
- Keep your mailboxes tidy