Concurrent Programming

CS511

About Erlang

- Functional language
- Concurrent/Distributed
  - No shared memory[1] (message passing)
- No types at compile time
  - Dynamically typed
- Open source
- Developed in the 80s in Ericsson by Joe Armstrong, Robert Virding and Mike Williams

[1] Except for process dictionaries

Runtime System

- Compiled code runs on a virtual machine (BEAM).
- Lightweight processes
- Fast process creation
- Support hot-swapping

More About Erlang

- Open Telecom Platform (OTP)
  - OTP is set of Erlang libraries and design principles providing middle-ware to develop these systems. It includes its own distributed database, applications to interface towards other languages, debugging and release handling tools.[2]
- Supporting tools
  - Debugger
  - Unit testing
  - Dialyzer
  - Mnesia

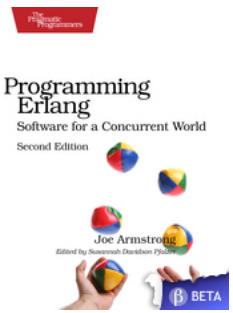
[2] https://www.erlang.org

Typical Applications

- Telecoms
  - Switches (POTS, ATM, IP, ...)
  - GPRS
  - SMS applications
- Internet applications
  - WhatsApp (backbone)
  - Facebook (chat)
  - Amazon (SimpleDB, part of the Amazon Elastic Compute Cloud)
  - Yahoo! (social bookmarking service, Delicious)
  - Online shopping (Klarna AB)
  - T-Mobile
  - Discord
- 3D modelling (Wings3D)

Bibliography

- Programming Erlang: Software for a Concurrent World, Joe Armstrong



- Learn some Erlang for Great Good!, Fred Hebert
- Erlang Programming, Cesarini and Thompson

Note: Some of the material in these slides draw from the above sources

Downloading and Installing

- http://www.erlang.org/downloads
- Prepackaged binaries from https://www.erlang-solutions.com/resources/download.html
- Setting up erlang mode in emacs+flycheck: http://www.lambdacat.com/post-modern-emacs-setup-for-erlang/

Erlang

Shell

Data Types

Modules

Running Erlang

```
$ erl
Erlang/OTP 19 [erts-8.0] [source-6dc93c1] [64-bit] [smp:4:4]
     [async-threads:10] [hipe] [kernel-poll:false]

Eshell V8.0  (abort with ^G)
1> io:format("Hello, world!~n").
Hello, world!
ok
2> q().
ok
$
```

A Small Script

```
$ cat hello.erl
%% Sample module
-module(hello).
-export([hello/0]).
hello() -> "Hello, world!".
$ erl
Erlang/OTP 19 [erts-8.0] [source-6dc93c1] [64-bit] [smp:4:4]
     [async-threads:10] [hipe] [kernel-poll:false]

Eshell V8.0  (abort with ^G)
1> c(hello).
{ok,hello}
2> hello:hello().
"Hello, world!"
3> q().
```

Mind the dot!

Sequential Fragment

- Data types and variables
- Function definitions
- Pattern matching

Numbers

Integers (arbitrarily big)

```
7> fact:fact(200).
788657867364790503552363213932185062295135977687173263294742
533244359449963403342920304284011984623904177212138919638830
257642790242637105061926624952829931113462857270763317237396
988943922445621451664240254033291864131227428294853277524242
407573903240321257405579568660226031904170324062351700858796
178922222789623703897374720000000000000000000000000000000000
00000000000000
```

Floats

```
> 2.78 + 9.6.
12.37999999999999
```

- IEEE 754 de 64–bits (range: $\pm 10^{308}$)

Atoms

```
1 start_with_a_lower_case_letter
2 'Anything_inside_quotes\n\09'
```

- Names must begin in lowercase or between apostrophes
- Heavily used

Characters and Strings

- Characters: `$a`, `$n`.
- Strings

```
9> "hello".
"hello"
```

- They are, in fact, a list of integers.

```
10> "hello\7".
[104,101,108,108,111,7]
11> is_list("hello").
true
```

Tuples

```
{}
{atom, another_atom, 'PPxT'}
{atom, {tup, 2}, {{tup}, element}}
{atom, {"hello",5}}

12> is_tuple({atom, another_atom, 'PPxT'}).
true
13> is_list({atom, another_atom, 'PPxT'}).
false
14> is_atom({atom, another_atom, 'PPxT'}).
false
```

Modeling Data

```
type 'a tree = Leaf of 'a | Node of ('a tree)*('a tree)
```

- Atoms to indicate which constructor is used at top-level
- Tuples to collect the arguments of the constructor
- Example:

```
Node(Node(Leaf 3,Leaf 4), Leaf 5)
```

becomes

```
{node, {node, {leaf, 3}, {leaf, 4}}, {leaf, 5}}
```

Operators

- Arithmetic: +, -, *, /, `div`, `rem`
- Equal value: "==" and "/="
- Exact equality (type and value):
  "=:=" and "=/="
- Boolean: `and`, `or`, `xor`, `not`, `andalso`, `orelse`

Operator Examples

```
1> 4 == 4.0.          % value is 4 on both sides
true
2> 4 =:= 4.0.         % value same but type different
false
3> 1 =/= 0.
true
4> 1 /= 1.
false
5> 1 =/= 1.0.
true.
```

- Use `==` and `/=`
- Switch to `=:=` and `=/=` only when you need exact equality (type and value)

Lists

```
1 []
2 [1, true]
3 [1 | [true] ]
4 [ok, 10]
```

- List concatenation: “++”
- List subtraction: “--”
- List cons: “|”
- List comprehension: `[ math:log(A)|| A <- lists:seq(1,10)]`

Operator and List Examples

```
3> L1 = [ apple, cherry ].
[apple,cherry]
4> L2 = [ lime, grape ].
[lime,grape]
5> L3 = L1 ++ L2.
[apple,cherry,lime,grape]
6> L3 -- [cherry].
[apple,lime,grape]
7> L4 = [ banana | L3 ].
[banana,apple,cherry,lime,grape]
8> [ Head | Tail ] = L4.
[banana,apple,cherry,lime,grape]
```

- Note: = is not assignment; it is matching

Operator and List Examples (cont.)

```
9> b().                    % b() shows all bindings
Head = banana              % Head and Tail have been bound
L3 = [apple,cherry,lime,grape]
L4 = [banana,apple,cherry,lime,grape]
Tail = [apple,cherry,lime,grape]
ok
10> f().                   % f() flushes all bindings
ok
11> { A, B } = { 4.0, 5.2 }.
{4.0,5.2}
12> b().
A = 4.0
B = 5.2
13> { C, D } = { 4.0, 5.2 }.
{4.0,5.2}
14> { A, B } == { C, D }.
true
15> { A, B } =:= { C, D }.
true
```

Comparison

- In Erlang all terms are comparable

$$number < atom < reference < fun < port < pid < tuple < map < nil < list < bitstring$$

- Integers and floats are compared as usual
- The rest are compared as indicated above

```
1> a<2.
false
2> 2<a.
true
```

More on Variables

- Identifiers: `A_long_variable_name`
- Must start with an upper case letter
- Can store values
- Can be bound only once!
- Bound variables cannot change values
- We use the $=$ operator for binding (and also matching!)

More on Variables

```
1> a = 3.                  % fails because a is not a variable
** exception error: no match of right hand side value 3
2> A = 3.                  % notice: ends with a period
3
3> B = 3.                  % there's that period again
3
4> A = B.                  % succeeds: A and B both have value 3
3
5> A = 4.                  % fails because A cannot be re-bound
** exception error: no match of right hand side value 4
6> X = { hello, goodbye }.     % hello & goodbye are atoms
{hello,goodbye}
7> { Y, Z } = X.           % binds both Y and Z
{hello,goodbye}
8> Y.
hello
9> Z.
goodbye
```

More on Variables

```
10> X = {Y,Z}.          % succeeds because of value match
{hello,goodbye}
11> X = {hello,Z}.      % succeeds because of value match
{hello,goodbye}
12> q()                 % notice: forgot the period
12> q().                % fails because Erlang reads "q()q()."
* 2: syntax error before: q
13> q().                % succeeds: quits Erlang shell
ok
```

Functions

- May have several clauses
  - Function is sequence of pattern matching clauses separated by semicolons – semicolon means "or"
  - Finish definition with .
- Function application matches arguments to pattern in some clause

```
1 fact(0) -> 1;
2 fact(N) when N>0 -> N * fact(N-1).
```

- "`when ...`" is a clause guard

Example 1

```
1 arith(X, Y) ->
2     io:format("Arguments: ~p ~p~n", [ X, Y ]) ,
3     Sum = X + Y ,
4     Diff = X - Y ,
5     Prod = X * Y ,
6     Quo = X div Y ,
7     io:fwrite("~p ~p ~p ~p~n", [ Sum, Diff, Prod, Quo ]) ,
8     { Sum, Diff, Prod, Quo } .
```

Take note:

- Function name starts with lowercase letter
- `io:format` is similar to `printf`
- Expressions separated by comma
- Function clause ended by period
- Final expression is function's return value

Example 2

```
1 what_day(saturday) -> % "saturday" is an atom
2     weekend ;         % notice semicolon
3 what_day(sunday) ->   % "sunday" is an atom
4     weekend ;         % semicolon again
5 what_day(_) ->        % underscore is "don't care" variable
6     weekday .         % period ends function
```

Example 3

```
1  drivers_license(Age) when Age < 16 ->
2      forbidden ;
3  drivers_license(Age) when Age == 16 ->
4      'learners permit' ;
5  drivers_license(Age) when Age == 17 ->
6      'probationary license' ;
7  drivers_license(Age) when Age >= 65 ->
8      'vision test recommended but not required' ;
9  drivers_license(_) ->
10     'full license'.
```

Function Application

- Function application is call-by-value or eager
- Clause matches if function name, arguments, and all guards match the input
- Except for "built-in functions (BIFs)" must specify function's module when calling

```
1 2> drivers_license(16).
2 ** exception error: undefined shell command
       drivers_license/1
3 3> example:drivers_license(16).
4 'learners permit'
```

Function Application

```
1> c(example).                    % c() compiles
{ok,example}
2> drivers_license(16).           % must specify module
** exception error: undefined shell command drivers_license
     /1
3> example:drivers_license(16).
'learners permit'
4> example:drivers_license(15).
forbidden
5> example:drivers_license(17).
'probationary license'
6> example:drivers_license(23).
'full license'
7> example:drivers_license(65).
'vision test recommended but not required'
8> q().
ok
```

Pattern Matching

The factorial definition uses pattern matching over numbers

```
1 fact(0) -> 1;
2 fact(N) when N>0 -> N * fact(N-1).
```

- A zero number (first clause)
- A number different from zero (second clause)

A more involved example. Function area to compute the area of different geometrical figures.

```
1 area({square, Side})  -> Side * Side ;
2 area({circle, Radius}) -> Radius*Radius*math:pi().
```

- Patterns: `{square, Side}` and `{circle, Radius}`
- `{square, Side}` matches `{square, 4}` and binds 4 to variable `Side`
- `{circle, Radius}` matches `{circle, 1}` and binds 1 to variable `Radius`

Anonymous Functions

```
11> F = fun (Y) -> Y+1 end.
#Fun<erl_eval.6.127694169>
12> F(2).
3
13> G = fun () -> 1 end.
#Fun<erl_eval.20.127694169>
14> G().
1
15> G.
#Fun<erl_eval.20.127694169>
16> [F,G].
[#Fun<erl_eval.6.127694169>,#Fun<erl_eval.20.127694169>]
```

Pattern Matching (cont.)

```
1 {B, C, D} = {10, foo, bar}
```

Succeeds: binds B to 10, C to foo and D to bar

```
1 {A, A, B} = {abc, abc, foo}
```

Succeeds: binds A to abc, B to foo

```
1 {A, A, B} = {abc, def, 123}
```

Fails

```
1 [A,B,C,D] = [1,2,3]
```

Fails

Pattern Matching (cont.)

```
1 [H|T]= [1,2,3,4]
```

Succeeds: binds H to 1, T to [2,3,4]

```
1 [H|T] = [abc]
```

Succeeds: binds H to abc, T to []

```
1 [H|T] = []
```

Fails

```
1 {A, _ , [ B | _ ] , {B} } = {abc,23,[22,x],{22}}
```

Succeeds: binds A to abc, B to 22

BIFs

- Much-used modules in Erlang library: `io`, `list`, `dict`, `sets`, `gb_trees`
- You can inspect the source code for these libraries
- Eg. snippet from /usr/local/lib/erlang/lib/stdlib-3.0/src/lists.erl

```
1 %% sum(L) returns the sum of the elements in L
2 -spec sum(List) -> number() when
3       List :: [number()].
4
5 sum(L)           -> sum(L, 0).
6
7 sum([H|T], Sum) -> sum(T, Sum + H);
8 sum([], Sum)     -> Sum.
```

Erlang

Shell

Data Types

Modules

Modules

- Basic compilation unit is a module
  - Module name = file name (.erl)
- They contain attributes and function definitions
- Attributes are of the form `-Name(Attribute).` and describe information about the module
- Let us create the module `math_examples` as follows.

```
1 -module(math_examples).
2 -export([fact/1,area/1]).
3 -author("E.B").
4
5 fact(0) -> 1;
6 fact(N) when N>0 -> N * fact(N-1).
7
8 area({square, Side})  -> Side*Side ;
9 area({circle, Radius}) -> Radius*Radius*math:pi().
```

Modules

Running the examples.

```
1> c(math_examples).
{ok,math_examples}
2> math_examples:fact(3).
6
3> math_examples:area({square,4}).
16
4> math_examples:area({circle,1}).
3.14159265358979 3
```

Modules

- A useful compiler option is `export_all`

```
5> c(a_module,[export_all]).
{ok,a_module}
```

- This causes the compiler to ignore the `-export` module attribute and export all functions defined
- You can also do the following:

```
-module(math_examples).
-compile(export_all).
-author("E.B").

fact(0) -> 1;
fact(N) when N>0 -> N * fact(N-1).

area({square, Side})  -> Side*Side ;
area({circle, Radius}) -> Radius*Radius*math:pi().
```

Modules – Information About

```
1> c(math_examples).
{ok,math_examples}
2> math_examples:module_info().
[{module,math_examples},
 {exports,[{fact,1},
           {area,1},
           {module_info,0},
           {module_info,1}]},
 {attributes,[{vsn,[292229879300425682399740783243125416564]
     },
              {author,"E.B"}]},
 {compile,[{options,[]},
           {version,"7.0"},
           {source,"/Users/ebonelli/Documents/erlang/
                math_examples.erl"}]},
 {native,false},
 {md5,<<219,217,109,113,225,135,117,96,156,42,192,248,50,
        41,98,116>>}]
```

Modules – Command Line

- Compilation

```
1 $ erlc math_examples.erl
```

  This generates a bytecode file (i.e. `.beam` file)

- Execution

```
1 $ erl -noshell -run math_examples factLstStr 5 -run
      init stop
```

  `factStr` consumes a list of strings and then prints the result by calling `fact`

Records

- Declared as module attributes

Definition

```
-module(records).
-compile(export_all).

-record(robot, {name, type=industrial, hobbies, details=[]})
    .
```

Creation

```
first_robot() ->
    #robot{name="Mechatron", type=handmade,
           details=["Moved by a small man inside"]}.
```

Records are Syntactic Sugar for Tuples

Accessing record fields

```
1> c(records).
{ok,records}
2> records:first_robot().
{robot,"Mechatron",handmade,undefined,["Moved by a small man
     inside"]}
```

- Note above: a record is just syntactic sugar for a tuple
- That means that if we try to access a field, we'll get an error

```
3> (records:first_robot())#robot.name.
* 1: record robot undefined
```

Accessing the Fields

We must load the record definitions first

```
3> rr(records).
[robot]
4> records:first_robot().
#robot{name = "Mechatron",type = handmade,hobbies =
    undefined, details = ["Moved by a small man inside"]}
```

```
16> (records:first_robot())#robot.name.
"Mechatron"
17> Crusher = #robot{name="Crusher", hobbies=["Crushing
    people","petting cats"]}.
#robot{name = "Crusher",type = industrial,
       hobbies = ["Crushing people","petting cats"],
       details = []}
18> Crusher#robot.name.
"Crusher"
```

Updating Records

```
repairman(Rob) -> Details = Rob#robot.details,
                  NewRob = Rob#robot{details=["Repaired by
repairman"|Details]}, {repaired, NewRob}.
```

```
16> c(records).
{ok,records}
17> records:repairman(#robot{name="Ulbert", hobbies=["trying
     to have feelings"]}).
{repaired,#robot{name = "Ulbert",type = industrial, hobbies
    = ["trying
to have feelings"], details = ["Repaired by repairman"]}}
```

Records in Header Files

```
1 %% this is a .hrl (header) file.
2 -record(included, {some_field,
3    some_default = "yeah!",
4    unimaginative_name}).
```

To include it in `records.erl`, add the following line to the module:

```
1 -include("records.hrl").
```

And an example function

```
1 included() -> #included{some_field="Some value"}.
```

Records in Header Files

```
18> c(records).
{ok,records}
19> rr(records).
[included,robot,user]
20> records:included().
#included{some_field = "Some value",some_default = "yeah!",
unimaginative_name = undefined}
```