Starting with Erlang
Sequential Programming in Erlang (Overview)

Walter Cazzola
Dipartimento di Informatica e Comunicazione
Università degli Studi di Milano
e-mail: cazzola@dico.unimi.it

Erlang
A Few of History

- 1981 — the Ericsson CS Lab has been founded
  1981-1986
  - a lot of work to decide which paradigm would be better to
    use in the telecommunication domain;
  - conclusions: doesn't exist the perfect paradigm but several
    characteristics should be mixed

- 1981 Erlang is born
  - the name is after the Danish mathematician Agner Krarup
    Erlang but could also mean Ericsson language

- 1987-1991
  - the JAM ("Joe's Abstract Machine") virtual machine (inspired
    by the Prolog WAM) has been implemented (in C);
  - in 1998 it has been replaced by BEAM ("Bogdan/Björn's Er-
    lang Abstract Machine")

- 1996 — Open Telecom Platform (OTP) has been released
  1998
  - Ericsson stops to develop Erlang but not to use it
  - Erlang becomes open source
  - since 2006 the BEAM supports multi-core processors

Erlang
Overview

Erlang is concurrency oriented, i.e., the process is the basic of
every computation.

Erlang adopts the actor's model for concurrency with
- asynchronous message exchange,
- non shared memory

Erlang is a dynamically typed functional language.

Erlang supports distribution, fault tolerance and hot-swapping
(dynamic SW updating).

My First Erlang Program
Again a Factorial!!!

-module(fact).
-export([fact/1]).

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

The program must be run through the BEAM shell

```
12:56 cazzola@surtr:~>erl
Erlang R13B04 (erts-5.7.5) [source] [64-bit] [smp:8:8] [rq:8] [async-threads:0] [kernel-poll:false]
Eshell V5.7.5 (abort with ^G)
1> c(fact).
{ok,fact}
2> fact:fact(7).
5040
3> fact:fact(100).
9332621544394415268169923885626670049071596826438162146859296389521759993229915608941463976156518
286253697920827223758251185210916864000000000000000000000000
```

Alternatively it could be run as a script via escript or through
native compilation via HiPE
Sequential Erlang Overview

Numbers and Atoms

1> 10. 10
2> 16#FF. 255
3> $A. 65
4> -12.35e-2. -0.1235

– decimal is used to store numbers in base “10”;
– #char is used for ascii values.

1> cazzola@dico.unimi.it. 'cazzola@dico.unimi.it'
2> 'Walter Cazzola'. 'Walter Cazzola'
3> 'Walter\nCazzola'. 'Walter
Cazzola'

– atoms start with lowercase letter but can contain any character;
– if quoted they can start by uppercase letters.

Sequential Erlang Overview

Tuples and Lists

1> {123, "walter", cazzola}. {123,"walter",cazzola}
2> {}. {}
3> {abc, {'Walter', 'Cazzola'}, 3.14}. {abc,{'Walter','Cazzola'},3.14}
4> {{1,2},3}=={1,{2,3}}. false

– used to store a fixed number of items;
– tuples of any size, type and complexity are allowed.

1> [ 1]. [ 1]
2> [1|[]]. [1]
3> [1|2|[]]. [1,2]
4> [1,2,[1]]. true
5> [ 1 | 2 , 3 | 4 | 5]. [1,2,3,4,5]

– used to store a variable number of items;
– lists are dynamically sized.

Sequential Erlang Overview

Assignments & Pattern Matching

1> A = 1. 1
2> A = 2. ** exception error: no match of right hand side value 2

– are just name bindings to values and cannot be modified;
– start with an uppercase letter and _ is an anonymous variable.

– the bindings are created via pattern matching.

3> \[B|L\]=[a,b,c]. [a,b,c]
4> {A,B,L}. {1,a,[b,c]}
5> {X,X}={B,B}. {a,a}
6> {Y,Y}={X,b}. ** exception error: no match of right hand side value a,b

Sequential Erlang Overview

Functions & Modules

\(name(pattern_1, pattern_2, ..., pattern_n) [when guard_1] \rightarrow body_1; \)
\(name(pattern_1, pattern_2, ..., pattern_n) [when guard_2] \rightarrow body_2; \)
\(name(pattern_1, pattern_2, ..., pattern_n) [when guard_1] \rightarrow body_4.\)

– clauses are scanned sequentially until a match is found;
– when a match is found all the variables in the head become bound,
– double can be called from outside the module, times is local to the module;
– double/1 means the function double with one argument (note that double/1 and double/2 are two different functions).
Sequential Erlang Overview

Guard Sequences

Each clause in function definition can be guarded by a guard sequence:
- a guard is a sequence $g_1, g_2, \ldots, g_n$ of guard expressions;
- a guard expression is a subset of Erlang expressions to guarantee to be free of side-effects;
- a guard sequence is true when all the guard expressions evaluate to true.

Valid guard expression are:
- the atom `true` and other constants;
- calls to some built-in functions (BIFs);
- arithmetic and boolean expressions; and
- short-circuit expressions (`andalso`/`orelse`).

Permitted BIFs are:
- `is_atom/1` is _binary/1 is _bitstring/1 is _float/1 is _function/2
- `is_function/1` is _integer/1 is _list/1 is _number/1 is _pid/1
- `is_port/1` is _record/2 is _record/3 is _reference/1 is _tuple/1
- `abs/1` `bit_size/1` `byte_size/1` `element/2` `float/1` `hd/1` `length/1` `node/0` `node/1` `round/1` `self/1` `size/1` `tl/1` `trunc/1` `tuple_size/1`}

Sequential Erlang Overview

Map, Filter & Reduce

- module(mfr).
- export([map/2, filter/2, reduce/2]).

map(_F, []) -> []; map(F, [H|TL]) -> [F(H)|map(F,TL)].

filter(true, P, H, L) -> [H|filter(P, L)]; filter(false, P, _, L) -> filter(P, L).
reduce(F, Q, []) -> Q; reduce(F, Q, [H|TL]) -> reduce(F, F(Q,H), TL).

1> mfr:map(fun(X) -> X*X end, [1,2,3,4,5,6,7]).
[1,4,9,16,25,36,49]
2> mfr:filter(fun(X) -> (X rem 2)==0 end, [1,2,3,4,5,6,7]).
[2,4,6]
3> mfr:reduce(fun(X,Y) -> X+Y end, [1,2,3,4,5,6,7]).
28

They are available in the module lists.

Sequential Erlang Overview

List Comprehensions

[Y|Qualifier, ..., Qualifier_n] is an arbitrary expression and each qualifier is either a generator or a filter:
- generators are in the form `Pattern <- ListExpr` where `ListExpr` evaluates to a list;
- filters are either predicates or boolean expressions.

- module(sort).
  - export([qsort/2]).

- module(prime).
  - export([primes/1]).
primes(N) when N>1 -> [X|| X <- lists:seq(2,N), (length(Y || Y <- lists:seq(2, trunc(math:sqrt(X))), (X rem Y) == 0)) == 0]; primes(_) -> [].

References

▶ Gul Agha. 
Actors: A Model of Concurrent Computation in Distributed Systems. 

▶ Joe Armstrong.
Programming Erlang: Software for a Concurrent World

▶ Francesco Cesarini and Simon Thompson.
Erlang Programming: A Concurrent Approach to Software Development
O’Reilly, June 2009.