Erlang 101

Google Doc

Erlang? with buzzwords

Erlang is a functional concurrency-oriented language with extremely low-weight user-space "processes", share-nothing message-passing semantics, built-in distribution, and a "crash and recover" philosophy proven by two decades of deployment on large soft-realtime production systems.

Source

Erlang?

- A programming language designed for concurrency, fault-tolerance and low-latency

- Academia meets Industry
- Rediscovered in 2006 (multi-core)
- Distribution is built-in (scalability)
- Not as much press as Go or Node.js

Who's using it?

- Facebook (Chat)
- Amazon (SimpleDB)
- Heroku (routing, logging)
- Github (RPC servers)
- T-Mobile, Motorola, Ericsson
- Basho (Riak)
- CouchDB
- WhatsApp
- RabbitMQ

History

Created in the late 80's at Ericsson's research lab, for programming telecommunication switching systems

More reading

Language requirements

- Handling a very large number of concurrent activities
- Actions to be performed within a certain time
- Systems distributed over several computers
- Interaction with hardware
- Very large software systems
- Complex functionality such as feature interaction
- Continuous operation over several years
- Software maintenance without stopping the system
- Stringent quality and reliability requirements
- Fault tolerance both to hardware failures and software errors

Sequential language

Functional language with single-assignment, dynamic typing and a Prolog-like syntax

Starting out

Variables

A variable *must start with a capital letter* (otherwise it's an atom)

```
x = 2. %% 'x' is not the same than 2
** exception error: no match of right hand side value 2
```

Single assignment

```
X = 1.
X = 2. %% 1 is not the same than 2
** exception error: no match of right hand side value 2
```

Pattern Matching

Variables are bound through pattern matching
(a generalization of "destructuring")

Pattern matching occurs with: =, case-of, function clauses

```
%% we start with X unbound
X = 5 %% now, X is bound to 5
X = 5 %% now, = behaves like assert
```

A match that fails is a runtime error

Pattern Matching (2)

```
%% Lists
[H|T] = [a, list]  %% H = a, T = [list]
T = [list|[]]

%% Tuples
{A, _, C} = {a, 3, tuple} %% A = a, C = tuple

%% Records
P = #person{name="Joe", age=34}
#person{name=[Initial|_]} = P  %% Initial = $J
```

Datatypes

```
Int = 100000000000000000000000000 % big ints
Float = 1.0e40
Char = $A
Atom1 = an_atom % starts with lowercase
Atom2 = 'It can also be quoted'
String1 = "a string"
String2 = [97,32,115,116,114,105,110,103]
Fun = fun() -> i_do_nothing end
Pid = spawn(Fun)
<<Int1:2, Int2:6>> = <<129>> % 10 000001 ~ 129
L = [a, list],
T = {a, tuple}
```

Binary

```
decode(Segment) ->
 case Segment of
   <<SourcePort:16, DestinationPort:16, SequenceNumber:32,
   AckNumber:32, DataOffset:4, _Reserved:4, Flags:8, WindowSize:16,
   Checksum:16, UrgentPointer:16, Payload/binary>> when DataOffset > 4 ->
      OptSize = (DataOffset - 5)*32,
      << Options:OptSize, Message/binary >> = Payload,
      <<CWR:1,ECE:1,URG:1,ACK:1,PSH:1,RST:1,SYN:1,FIN:1>> = <<Flags:8>>,
        %% Can now process the Message
       do_something_with(Message);
    _ ->
        {error, bad_segment}
end.
```

Booleans

No boolean type, just the atoms true and false.

Operators:

```
<, >, =<, >=
1 == 1.0 , 1 /= a
a =:= a, 1 =/= 1.0  (also checks type)
and, or
andalso, orelse (short-circuit)
not
```

Records (structs)

Records are syntactic sugar (they are compiled to tuples)

```
-define(person, {name, age}). %% definition

P = #person{name="Joe", age=34}
%% compiles to {person, "Joe", 34}

Age = P#person.age      %% field access
#person{age=Age} = P    %% same with PM

P2 = P#person{age=35}  %% record update
```

Functions are first-class

```
%% Double closes over Factor
Factor = 2,
Double = fun(X) -> Factor * X end

%% function as argument and a result
derive(F) ->
    fun(X) -> (F(X+0.001) - F(X)) / 0.001 end.

Two = derive() %% Two is a function

Two(whatever) %% 1.9999999999997797
```

Functions

No nested scope, once a variable is bound, it refers to the same value throughout the function (like Javascript, unlike C)

```
F = fun(Y) ->
 X = 5,
 case Y of
   X -> io:format("Y = 5~n"); % X is not fresh
   _ -> io:format("Y /= 5~n")
 end.
```

Case expressions (switch)

```
FizzBuzz = case {N rem 3, N rem 5} of
               {0,0} -> "fizzbuzz";
               {0,_} -> "fizz";
               {_,0} -> "buz";
                _    -> ok
           end
```

Clause order matters

They are checked from top to bottom

case-of is a generalization of *if-then-else*

Loops

No loop construct, use recursion

```
repeat_hello(0) -> done;
repeat_hello(N) ->
    io:format("Hello~n"),
    repeat_hello(N-1). %% no stack overflow

lists:foreach(
  fun(_) -> io:format("Hello~n") end,
  lists:seq(1,N)). %% also map, filter, foldl...
```

Modules

- In a file named after the module
- Contains only function definitions
- Only exported functions are visible from outside
- Calls from outside, must be qualified with the module name:

```
lists:seq(1,5).
```

Concurrency

related to the Actor model:

- lightweight processes
- each process has an id and a mailbox (unbounded)
- They communicate by sending asynchronous messages
- which they retrieve from their mailbox
- and then, can create other processes
- and change their internal state

Hitchhiker's guide

Concurrent language

```
%% spawn a new process to run loop()
Pid = spawn(fun() -> loop(0) end)
%% send Message to Pid
Pid ! Message

%% receive messages
loop(Count) ->
  receive %% blocks until a msg matches !
    {ping, From} -> From ! pong,
                    loop(Count+1);
    stop -> Count
  end.
%% see also register and link
```

Erlang Runtime

Provides its own OS (cheap concurrency and error detection).

- (OS-level) thread pool for async I/O
- one scheduler per core
- scheduling is preemptive
- process isolation (separate heaps)
- process garbage-collected individually

Distribution

- Communication occurs between "nodes" (instances of the Erlang runtime)
- *Transparent*: talking to remote processes is like talking to local processes
- Assumes a safe network

Distribunomicon

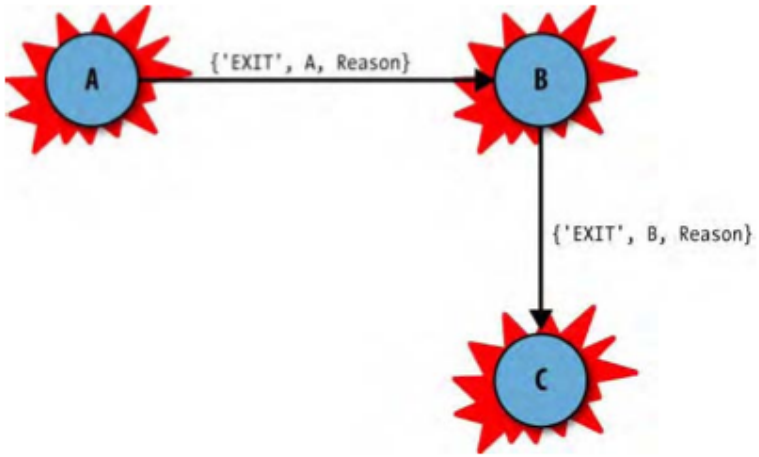
Fault-tolerance

The ability to recover from software and hardware errors.

This is why we need:

- distribution (can't be fault-tolerant with one machine)
- concurrency (for supervision)

Links



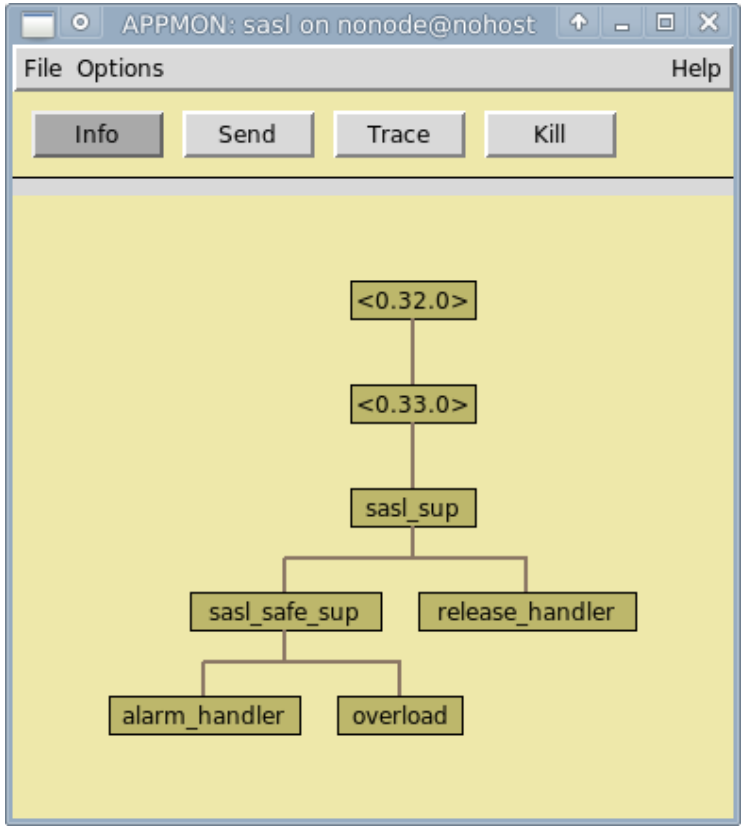
- Processes can be linked.
- When a process fails, its exit signal propagates to its 'link set', causing them to fail too.
- Mutually dependent processes should be linked.
- A process can choose to trap these signals and act upon them (supervisor).

Supervision trees

A way to organize processes in an app:

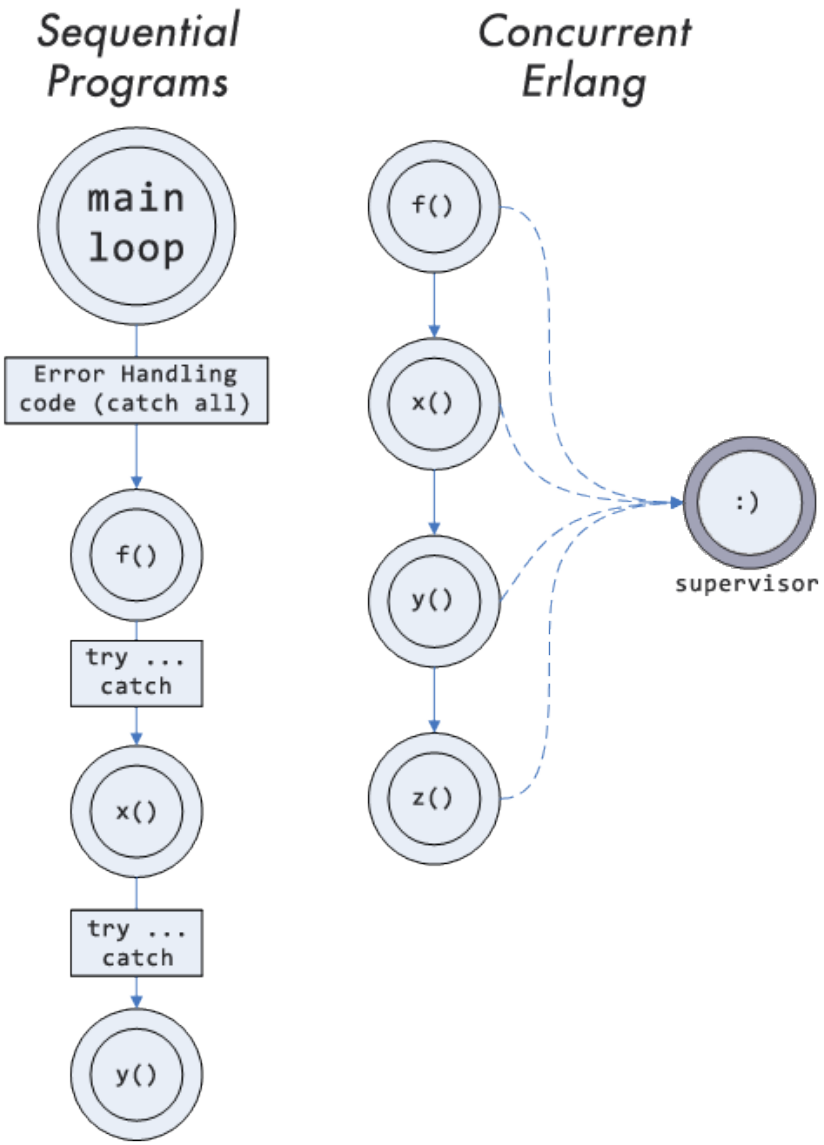
- nodes are supervisors,
- leaves are workers

Allow to monitor and restart subparts of the system



"Let it crash"

Instead of raising an exception, let the process crash and let another process (a supervisor) do the recovery.



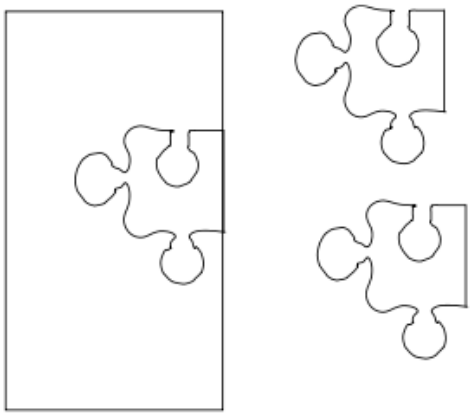
Open Telecom Platform

1. Conventions and good practices to guide the development, deployment and maintenance of Erlang applications

2. Libraries (behaviours) that abstract common programming patterns: servers, state machines, etc…

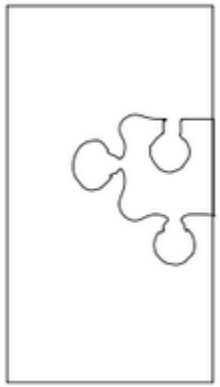
OTP behaviours

application: to package an application
supervisor: to build supervisor trees
gen_server: to build servers
gen_fsm: to build state machines
gen_event: to build event managers



gen_server

"Empty" server from which specific instances can be built.



Expects certain callbacks to be implemented:

- `init/1` state initialization
- `handle_call/3` synchronous request
- `handle_cast/2` asynchronous request
- `handle_info/2` out-of-band communication
- `terminate/2` state cleanup
- `change_code/3` hot-code update

Where next?

Code used in the talk
https://github.com/voila/Erlang101

Online Tutorial
http://www.tryerlang.org/

Books
Learn you some Erlang great coverage and available online, recommended !!!
Erlang Programming (O'Reilly)
OTP in action

Docs
http://erldocs.com/
http://erlang.org/doc/

Community
http://erlangcentral.org/
http://erlang.org/pipermail/erlang-questions/

If everything else failed…

Erlang: The Movie