# Erlang - functional programming in a concurrent world

Johan Montelius

KTH

HT15



## Erlang

Concurrent Oriented Programming

- processes have state
- communicate using message passing
- access and location transparent
- asynchronous

Functional programming

- evaluation of expressions
- recursion
- data structures are immutable



## History

Developed at Ericsson in late eighties, early nineties.

Targeting robust applications in the telecom world.

Survived despite "everything must be Java"

Growing interest from outside Ericsson.



## Today






## Why Erlang?

- concurrency built-in
- multicore performance
- simple to implement fault tolerance
- scales well in distributed systems



## Erlang

- the functional subset
- concurrency
- distribution
- failure detection



## Data structures

- Literals
  - atoms: foo, bar, ...
  - numbers: 123, 1.23 ..
  - bool: true, false
- Compound structures
  - tuples: {foo, 12, {bar, zot}}
  - lists: [], [1,2,foo,bar]



## Variables

- lexically scoped
- implicit scoping - the procedure definition
- untyped - assigned a value when introduced
- syntax: X, Foo, BarZot, _anything

Assignment of values to variables is done by pattern matching:

<Pattern> = <Expression>

A pattern can be a single variable:

- Foo = 5
- Bar = {foo, zot, 42}

or a compound pattern

- {A, B} = {4, 5}
- {A, {B, C}} = {41, {foo, bar}}
- {A, {B, A}} = {41, {foo, 41}}



## Patern matching

Pattern matching is used to extract elements from a datastructure.

{person, Name, Age} = find_person(Id, Employes),



## Pattern matching

Pattern matching can fail:

{person, Name, Age} = {dog, pluto}



## no circular structures

You can not construct circular data structures in Erlang.

Pros - makes the implementation easier.

Cons - I like circular structures.

## definitions

```
area(X, Y) -> X * Y.
```



## if statement

```
fac(N) ->
        if
           N == 0 -> 1;
           N > 0 -> N*fac(N-1)
        end.
```



## case statement

```
sum(L) ->
        case L of
           [] ->
                0;
           [H|T] ->
                H + sum(T)
        end.
```



## case statement

```
member(X,L) ->
        case L of
           [] ->
                no;
           [X|_] ->
                yes;
           [_|T] ->
                member(X, T)
        end.
```

## higher order

```
F = fun(X) -> X + 1 end.

F(5)
```



## higher order

```
map(Fun, List) ->
        case List of
             []  ->
               [];
          [H|T] ->
               [Fun(H) | map(Fun, T)]
     end.
```



## modules

```
-module(lst).
-export([reverse/1]).

reverse(L) ->
        reverse(L,[]).

reverse(L, A) ->
        case L of
           [] ->
              A;
           [H|T] ->
             reverse(T,[H|A])
        end.
```



## modules

```
-module(test).
-export([palindrome/1]).

palindrome(X) ->
        case lst:reverse(X) of
                X ->
                   yes;
                _ ->
                   no
        end.
```

## Concurrency

Concurrency is explicitly controlled by creation (spawning) of processes.

A process is when created, given a function to evaluate.

- no one cares about the result

Sending and receiving messages is the only way to communicate with a process.

- no shared state (. . . well, almost)



## spawn a process

```
-module(account)

start(Balance) ->
   spawn(fun() -> server(Balance) end).

server(Balance) ->
    :
    :
    :
```



## receiving a message

```
server(Balance) ->
        receive
           {deposit, X} ->
                server(Balance+X);

           {withdraw, X} ->
                server(Balance-X);

           quit ->
                ok
     end.
```



## sending messages

```
    :
Account = account:start(40),
Account ! {deposit, 100},
Account ! {withdraw, 50},
    :
```

## rpc-like communication

```
server(Balance) ->
   receive
        :
   {check, Client} ->
            Client ! {saldo, Balance},
            server(Balance);
        :
   end.
```



## rpc-like communication

```
friday(Account) ->
    Account ! {check,  self()},
    receive
        {saldo, Balance} ->
            if
              Balance > 100 ->
                    party(Account);
              true ->
                   work(Account)
            end
     end.
```



## implicit deferral

A process will have an ordered sequence of received messages.

The first message that matches one of several program defined patterns will be delivered.

Pros and cons:

- one can select which messages to handle first
- risk of forgetting messages that are left in a growing queue



## registration

A node register associate names to process identifiers.

```
register(alarm_process, Pid)
```

Knowing the registered name of a process you can look-up the process identifier.

*The register is a shared data structure!*

## registration

Erlang nodes (an Erlang virtual machine) can be connected in a group .

Each node has a unique name.

Processes in one node can send messages to and receive messages from processes in other nodes using the same language constructs



## starting a node

```
moon> erl -sname gold -setcookie xxxx
 :
 :
(gold@moon)>
```



## failure detection

- a process can monitor another process
- if the process dies a messages is placed in the message queue
- the message will indicate if the termination was normal or abnormal or ..... if the communication was lost



## monitor

```
Ref = erlang:monitor(process, Account),
Account !  {check, self()},


receive
  {saldo, Balance} ->
              :

  {'DOWN', Ref, process, Account, Reason}->
            :

end
```

## automatic termination

- A process can link to another process, if the process dies with an exception the linked process will die with the same exception.
- Processes that depend on each other are often linked together, if one dies they all die.



## linking

```
P = spawn_link(fun()-> server(Balance)  end),
do_something(P),
```



## Summary

- functional programming
- processes
- message passing
- distribution
- monitor/linking