Erlang - functional programming in a concurrent world

Johan Montelius
KTH

Erlang

- Concurrent Oriented Programming
  - processes have state
  - communicate using message passing
  - access and location transparent
  - asynchronous
- Functional Programming
  - 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

- Robust server applications:
  - Amazon SimpleDB - database
  - GitHub - code repository
  - RabitMQ - messaging middleware
  - Facebook - chat
- Why
  - simple to implement fault tolerance
  - scales well in distributed systems
  - multicore performance

Erlang

- background
- the functional subset
- concurrency
- distribution
- failure detection

Data structures

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

Variables

- lexically scoped
  - implicit scoping
  - procedure definition
  - case statement
- untyped
  - assigned a value when introduced
- Syntax
  - X, Foo, BarZot, _

Assignment and pattern matching

- 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 | T]} = {41, [foo, bar, zot]}

Pattern matching can fail

- Pattern matching is also used to do test and to guide the execution:
  - {person, Name, Number} = {dog, pluto}

No circular structures :-(

- You can not construct circular data structures in Erlang.
- Pros
  - makes the implementation easier
- Cons
  - I like circular structures.

Function 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.
```

Pattern matching


```
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)

spawning a process


```
-module(account)

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


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

Receiving messages


```
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},
    :
```

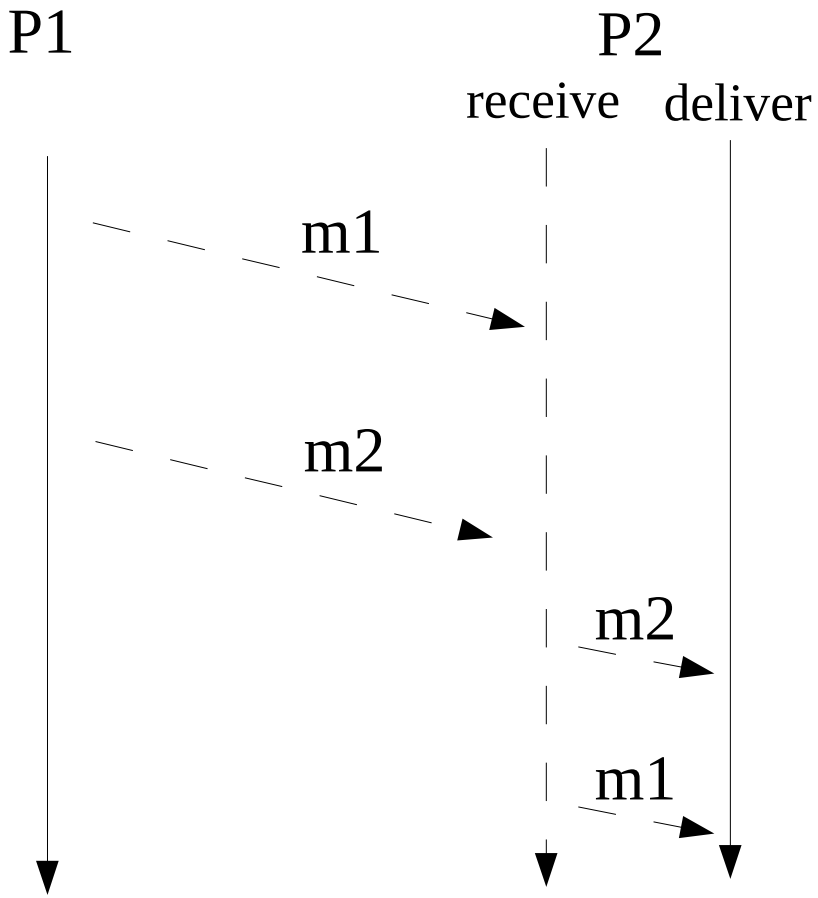
rpc-like communication


```
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.
```

Process communication



messages received in FIFO order

asynchronous sending, no acknowledgment

implicit deferral of message delivery

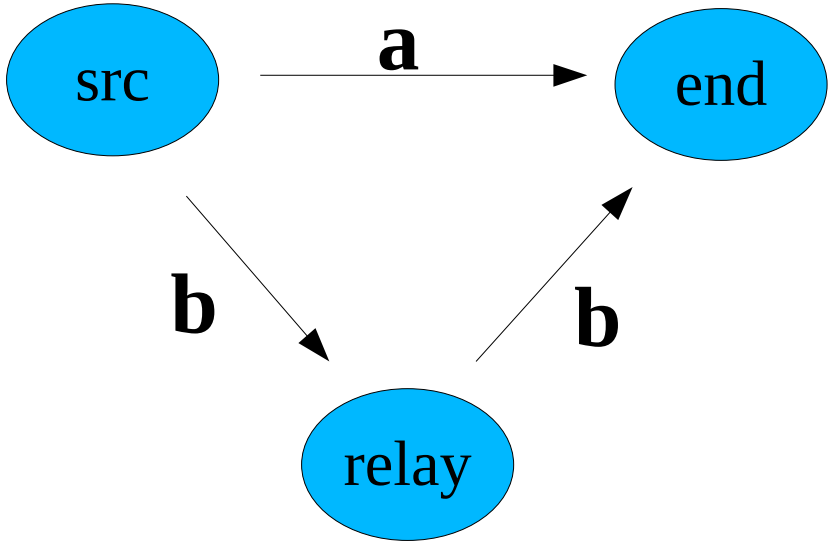
Implicit deferral

- 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

order of messages


```
src(End, Relay) ->
    End ! a,
    Relay ! b.

relay(End) ->
    receive
       X -> End ! X
    end.
```



Registration

- A *node register* associate names to process identifiers.
- Knowing the registered name of a process you can look-up the process identifier.
- The register is a shared data structure!

the name is the key

```
    :
MyAccount = account:start(400),
register(johan, MyAccount),
    :

          :
          if
               Balance > 100 ->
                    party(Account);
               true ->
                    johan ! {withdraw, 100),
                    Account ! {deposit, 100},
                    party(Account)
          end
```

Distribution

- 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 jhgsyt
:
:
(gold@moon)>
```

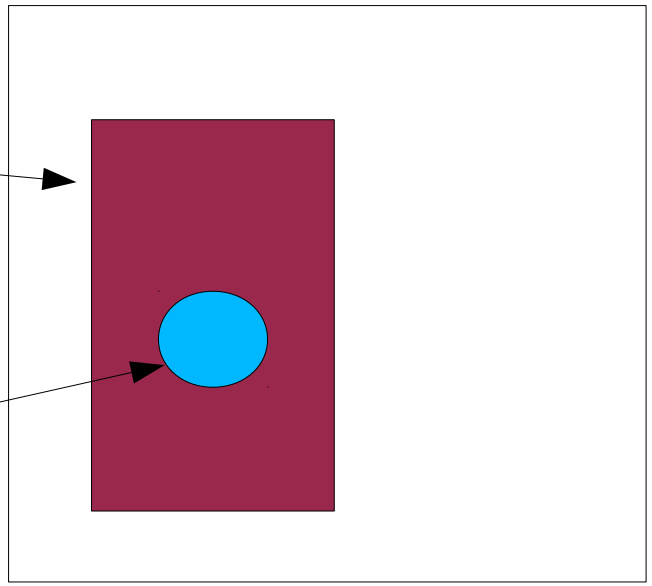
node registry

global name **{account, 'gold@moon'}**

a host called **moon**

a node called **gold**

a process registered as **account**



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 the account


```
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),
```

Erlang

- the functional subset
- concurrency
- distribution
- failure detection