A Taste of Erlang
DRAFT

D. Vermeir

December 2, 2009

What is Erlang?

> *Haskell is a **declarative functional** programming language with support for **concurrency**, **distribution**, **fault-tolerance***

declarative
: E.g. variables are as in Prolog, 'assign/unify once'.

functional
: No state

concurrency
: Communicating processes (millions of them).

distribution
: Processes can run on different machines

fault tolerance
: Using supervisors, erronuous processes can be automatically restarted, using a user-defined strategy.

Modules, Functions, Matching, Guards

```
-module(example1).
-export([factorial/1]).

% factorial/1 is interface of this module
factorial(N) when is_integer(N), N >0 -> factorial(N,1).

% internal factorial/2 uses accumulator
factorial(1, Accumulator) -> Accumulator.
factorial(N, Accumulator) when N>1 ->
        factorial(N-1, Accumulator*N).
```

- Variable names start with upper case, atoms much as in Datalog
- Function definition has clauses, matching of expressions causes variables to be bound
- First matching clause is executed, subject to **guards.**
- Last expression after `->` is returned.

Lists, List Comprehension

```
-module(example2).
-export([qsort/1]).

qsort([]) -> [].
qsort([Pivot| Rest]) ->
        qsort([X || X <- Rest, X < Pivot])
        ++ Pivot
        ++ qsort([ X || X <- Rest, X >= Pivot]).
```

- Prolog syntax
- List comprehension: `[X || X <- OtherList, ExtraConditions]`

Tuples, Records

```
-module(binary_tree).
-export([member/2]).

% member(Thing,Tree): does Thing appear in Tree
member(_, empty) -> false.
member(X, { _, X, _ } ) -> true.
member(X, {Left, Y, _ } ) when X < Y -> member(X, Left).
member(X, {_, Y, Right } ) when Y < X -> member(X, Right).
```

- Tuple (vector) `{A, b, C, d}`
- There are also records, but these are mapped internally to tuples

Higher Order Functions

```
-module(example3).
-export([foldl/3]).

% foldl(Function, InitialValue, List) applies Function
%  accross the list, starting with InitialValue, i.e.
%  foldl(F, V, [A,B,C]) = F(C, F(B, F(A, V) ) )
foldl(_, Accumulator, []) -> Accumulator.
foldl(F, Accumulator, [X | Rest]) ->
        NewAccumulator = F(X, Accumulator),
        foldl(F, NewAccumulator, Rest).
```

- The first argument of `foldl/3` is a function, making it a higher order function
- Anonymous functions may be defined using `fun(Args) ->.. end`, as shown below

```
-module(example4).
% Return the sum of a list of numbers
sum(Numbers) ->
  example3:foldl(fun(N, Total) -> N + Total end, 0, Numbers).
```

Communicating Processes (1/2)

```
-module(seq1).
-export([make_sequence/0, get_next/1, reset/1]).

make_sequence -> spawn(fun() -> loop(0) end).

loop(N) -> % tail recursion, constant space
  receive % From is PID of sender process to reply to
    { From, get_next } -> From ! { self(), N }, loop(N+1);
    reset -> loop(0)
  end.
```

- `spawn` starts new process executing its argument function and returns its PID (Process IDentifier).
- `receive` gets request from the mailbox, blocks if none of the available messages matches any of the patterns
- `PID!data` sends data to mailbox of process with PID.
- `self()` returns own PID
- `register(some_atom, Pid)` may be used to associate an atom with a process ('well known name').

Communicating Processes (2/2)

```
-module(seq1).
-export([make_sequence/0, get_next/1, reset/1]).
% client interface
get_next(SequenceProcess) ->
  SequenceProcess ! { self(), get_next },
  receive
    { SequenceProcess, N } -> N
  end.

reset(SequenceProcess) -> SequenceProcess ! reset.
```

Client Fragment

```
  SequenceProcess = seq1:make_sequence(),
  seq1:get_next(SequenceProcess), % 0
  seq1:get_next(SequenceProcess), % 1
  seq1:reset(SequenceProcess).
```

Abstracting Protocols: a Server Behaviour

```
-module(server).
-export([start/1, , call/2, cast/2]).
start(Module) -> spawn(fun() -> loop(Module, Module:init()) end).
loop(Module, State) ->
  receive
    { call, { Client, Id }, Params } ->
      { Reply, NewState } = Module:handle_call(Params, State),
      Client ! { Id, Reply },
      loop(Module, NewState);
    { cast, Params } ->
      NewState = Module:handle_cast(Params, State),
      loop(Module, NewState)
  end.
call(Server, Params) ->
  MsgId = make_ref(), % create unique ID
  Server ! { call, { self(), MsgId } , Params },
  receive
    { MsgId, Reply } -> Reply
  end.
cast(Server, Params) ->
  Server ! { cast, Params }.
```

Server Behaviour Callbacks

```
-module(server).
-export([start/1, , call/2, cast/2]).
start(Module) -> spawn(fun() -> loop(Module, Module:init()) end).
loop(Module, State) ->
  receive
    { call, { Client, Id }, Params } ->
      { Reply, NewState } = Module:handle_call(Params, State),
      Client ! { Id, Reply },
      loop(Module, NewState);
    { cast, Params } ->
      NewState = Module:handle_cast(Params, State),
      loop(Module, NewState)
  end.
..
```

The Callback Module Should Implement

```
%   init -> InitialState
%   handle_call(Params, NewState) -> { Reply, NewState }
%   handle_cast(Params, NewState) -> NewState
```

Using a Behaviour: forget concurrency

```
-module(seq2).
-export([make_sequence/0, get_next/1, reset/1]).
-export([init/0, handle_call/2, handle_cast/2]).
% seq2 API
make_sequence() -> server:start(seq2).
get_next(SeqServer) -> server:call(SeqServer, get_next).
reset_next(SeqServer) -> server:cast(SeqServer, reset).
% server callbacks
init() -> 0.
handle_call(get_next, N) -> { N, N+1 }.
handle_call(reset, _) -> 0.

% unit test: return 'OK' or throw exception
test() ->
  0 = init(),
  { 6, 7 } = handle_call(get_next, 6),
  0 = handle_cast(reset, 101),
  ok.
```

Example Standard (OTP) Behaviours

Generic Server

`gen_server` generalizes request/response pattern from client/server, RPC. Adds timeouts, delegation by server to another process, monitoring of server by client (immediately notified of server failure).

Generic Finite State Machine

`gen_fsm` clients signal events to the fsm, possibly waiting for reply

Generic Event Handler

`gen_event` dispatches received events to dynamically managed event handlers. Several specialisations are available.

Parallellism

```
% Calls = [ { Server, Params } .. ]
multicall1(Calls) ->
  Ids = [ send_call(Call) || Call <- Calls ],
  collect_replies(Ids).

send_call({ Server, Params }) ->
  Id = make_ref(), % generates unique ID
  Server ! { call, { self(), Id}, Params },
  Id.

collect_replies(Ids) ->
 [ receive { Id, Result } -> Result end || Id <- Ids ].
```

- Each request is identified with a unique ID
- **multicall1** stuffs server's mailbox with requests.
- **collect_replies** will wait for each reply in turn

⇒ **multicall1** blocks until answers have been obtained

More Parallellism using Worker Processes

```
multicall2(Calls) -> % Calls = [ { Server, Params } .. ]
  Parent = self(),
  Pids = [ worker(Parent, Call) || Call <- Calls ],
  % do something else
  wait_all(Pids).

worker(Parent, {Server, Params }) -> % new worker process
  spawn(fun() ->
          Result = server:call(Server, Params),
          Parent ! { self(), Result }
          end ).

wait_all(Pids) ->
 [ receive { Pid, Result } -> Result end || Pid <- Pids ].
```

- Process creation is cheap (less than 1microsecond)
- Processes are small (less than 1KBytes), you can have millions running at the same time.
- Processes can run on different machines (distribution)

Timeouts and signals

- Timeouts can be handled by a `receive` clause.
- A run-time error or a call to `exit(Reason)` cause an **abnormal exit** of the process.
- Processes can be linked using `link(Pid)` and then receive signal with `Pid` and `Reason` if `Pid` exits. By default, normal exit signals are ignored, abnormal exit signals cause an abnormal exit.

```
process(..) ->
  process_flag(trap_exit, true), % turns signals into messages
  ..
  link(From), % link to process with PID 'From'
  receive
    .. ;
    {'EXIT', From, Reason} -> % process with PID 'From' exited
        handle_exit(From, Reason),
    ..
    after 5000 -> % timeout of 5 secs reached
    end
  ..
```

Supervisors

- As **supervisor** spawns a set of children and links to them.
- It can use a strategy to restart failed children.
- Children can themselves be supervisors: tree structure.
- Linking is bidirectional, so 'orphaned' child processes may kill themselves if the supervisor dies.

References

- `http://www.erlang.org/` (Erlang official site)
- `http://www.trapexit.org/` (Erlang community)
- Most of the material on these slides comes from "*Erlang for Concurrent Programming*" by Jim Larson, CACM March 2009.