

Erlang

Charles Calkins

History

- Mid '80s – Ericsson finds existing languages deficient for development of new telecom systems
- Late '80s – 4 years spent developing a new language, Erlang, and prototyping applications in it. Prolog-based VM.
- 1991 - C-based VM written.
- 1994 - First commercial system implemented in Erlang
- 1995 – Erlang release mature enough to be used in multiple projects
- 1996 – OTP released
- Dec 1998 – Open-sourced

Variables

- Must start with an uppercase letter
`x, Bob, AVariable`
- Single assignment
- `=` is a pattern match operator
 - Acts like assignment if the variable is unbound
 - Match LHS = RHS if LHS is bound

Types...

- Numbers
 - Arbitrary length integers
 - Floating point
 - Atoms
 - Non-numerical constants
 - Begin with a lowercase letter or single quotation mark
- `fred, 'AnAtom`

...Types...

- Tuples
 - Delimited by { }
 - Elements separated by commas

```
P = { 1, 5.7, fred }
```

```
{ x, y, z } = P
```

X is bound to 1, Y to 5.7, Z to fred

```
{ 1, y, fred } = P
```

Only binds Y if others match, else badmatch error

...Types...

- Lists
 - Delimited by []
 - Elements separated by commas

```
L = [ 4, {bob, alice}, 9.1 ]
```

...Types...

- Strings are lists of character values
 - ...but can be characters surrounded by double quotes
 - Latin-1 (ISO 8859-1) encoding

```
s1 = "Fred"
```

```
s2 = [70, 114, 101, 100]
```

S1 = S2 succeeds

...Types

- Binaries are sequences of bytes
 - Delimited by << >>
 - Each element is an integer 0..255

```
Flag1 = 10.    Flag2 = 20.    Flag3 = 30.  
Bits16 = << Flag1:4, Flag2:6, Flag3:6 >>.  
<<165,30>>
```

(Think protocol parsing, e.g., IP header)

Functions

- One or more clauses
 - Separated by a semicolon
 - Last is terminated by a dot and whitespace
- Each clause:
 - Head
 - Function name
 - Pattern in parenthesis
 - Body
 - Sequence of expressions executed on a successful pattern match
 - Head and body separated by an arrow
- Function return is the evaluation of the body of the first clause with a pattern that matches

...Functions

perimeter() function, two clauses:

```
perimeter({square, W, H}) -> 2*W + 2*H;
```

```
perimeter({circle, R}) -> 2*3.14159*R.
```

Source Code

- Declare functions in modules (not in the shell)
 - Files have .erl extension
 - First lines:
 - `-module(name) .`
 - `-export([list of functions]) .`
- Compile with erlc, or `c(name)` in shell
 - Creates .beam file
 - Place in current directory or Erlang search path

Function in a Module

```
% basic.erl
-module(basic).
-export([perimeter/1]).  % /1 is function arity

perimeter({square, W, H}) -> 2*W + 2*H;
perimeter({circle, R}) -> 2*3.14159*R.
```

Interactive Shell

```
C:\>erlc basic.erl
```

```
C:\>erl
```

```
Eshell V5.8.2 (abort with ^G)
```

```
1> perimeter({circle, 5}).
```

```
** exception error: undefined shell command  
    perimeter/1
```

```
2> basic:perimeter({circle, 5}).
```

```
31.4159
```

```
3> basic:perimeter({square, 2, 4}).
```

```
12
```

```
4>
```

Anonymous Functions

```
% add to basic.erl  
doubler() -> fun(X) -> 2*X end.
```

```
C:\>erl  
Eshell V5.8.2 (abort with ^G)  
1> F = basic:doubler().  
#Fun<basic.0.471249>  
2> F(10).  
20  
3> lists:map(F, [5, 10, 15]). % function as argument  
[10,20,30]
```

List Processing

- Separate a list into a head element and tail list with `|`, recurse, processing each part

```
% add to basic.erl
total([Head|Tail]) -> Head + total(Tail);
total([]) -> 0.
```

```
C:\>erl
Eshell V5.8.2  (abort with ^G)
1> basic:total([1,2,3,4,5]).
15
```

List Comprehension...

- [Exp || Qual1, Qual2, ...]
 - Exp is an arbitrary expression
 - Qual n is a generator
 - Pattern <- ListExpression
 - ... or a filter
 - Predicate (function returning true or false)
 - Boolean expression
- Result is a list

...List Comprehension...

```
[ 3*x || x <- [1,2,3] ].  
[3,6,9]
```

```
[ x || {fred, x} <- [{fred, 7}, {bob, 8}, {fred, 2},  
  jane}].  
[7,2]
```

Map can be implemented this way, as it is a
function evaluated over each element of a list

```
map(F, L) -> [F(x) || x <- L].
```

...List Comprehensions...

- Define a predicate:

```
is_odd(X) -> X rem 2 == 1.
```

```
basic:is_odd(5).
```

```
true
```

```
basic:is_odd(4).
```

```
false
```

...List Comprehensions...

- Use it as a guard:

```
odds(N) -> [ x || x <- lists:seq(1,N), is_odd(x) ].
```

```
basic:odds(10).
```

```
[1,3,5,7,9]
```

...List Comprehensions...

- `filter()` selects elements from a list where the predicate is true:

```
odds2(N) -> lists:filter(fun(X) -> is_odd(X) end,  
    lists:seq(1,N)).
```

```
basic:odds2(10).
```

```
[1,3,5,7,9]
```

...List Comprehensions

Quicksort, with list comprehensions and guards:

```
quicksort([]) -> [];  
quicksort([P|T]) ->  
    quicksort([X || X <- T, X < P])  
    ++ [P] ++  
    quicksort([X || X <- T, X >= P]).
```

```
basic:quicksort([10,5,3,7,9]).  
[3,5,7,9,10]
```

Processes...

- Creation and destruction is fast
- Can have a large number of processes in each node
 - 32768 is the default, increase with +P command-line option
 - Return current limit:

```
erlang:system_info(process_limit).
```

- Communication is by message passing

...Processes...

- Many ways to create a process, e.g.:
 - To evaluate function `Fun` on the local node:
`Pid = spawn(Fun).`
 - To evaluate `Fun` on a remote node:
`Pid = spawn(Node, Fun).`
 - Given a module, function name and list of arguments (MFA):
`Pid = spawn(Mod, FuncName, ArgList).`
`Pid = spawn(Node, Mod, FuncName, ArgList).`
Invokes `Mod:FuncName(Arg1, Arg2, ...)`

...Processes...

- To send a message to a process (mailbox):

```
Pid ! Message
```

- To receive a message from a process:

```
receive
```

```
    Pattern1 [when Guard1] -> Exprs1;
```

```
    Pattern2 [when Guard2] -> Exprs2;
```

```
...
```

```
after MSDelay -> Exprs;
```

```
end
```


...Processes...

```
% use anonymous function to wrap tick(), else tick()  
% is evaluated (in the current process) and its  
% result passed to spawn()  
start_tick() -> spawn(fun() -> tick(0) end).
```

```
% tail recursion, passing state  
tick(Iter) ->  
  io:format("Tick ~p~n", [Iter]),  
  receive  
    { quit } -> io:format("Stopping~n")  
  after 5000 -> tick(Iter+1) % 5 second timeout  
end.
```

...Processes

```
C:\>erl
Eshell V5.8.2  (abort with ^G)
1> Pid = basic:start_tick().
Tick 0
<0.32.0>
2> Tick 1
2> Tick 2
2> Pid ! { quit }.
Stopping
{quit}
3>
```

Linked Processes...

- If a linked process dies, an exit signal is broadcast to all processes that it is linked to. To link:

```
link(Pid)
```

```
Pid = spawn_link(...)
```

- The linked processes will also die unless is set to be a system process to trap exits

```
process_flag(trap_exit, true).
```

- Receive a message when a linked process dies

```
{ 'EXIT', Pid, Why }
```

...Linked Processes...

```
-module(link).  
-export([local/1]).
```

```
local([RemoteHost, WhichTick]) ->  
    process_flag(trap_exit, true),  
    spawn_link(  
        list_to_atom(RemoteHost),  
        fun() ->  
            select_tick(list_to_integer(WhichTick))  
        end),  
    io:format("Exited with status: ~p~n",  
        [loop(list_to_integer(WhichTick))]),  
    init:stop().
```

...Linked Processes...

```
select_tick(WhichTick) ->  
  case WhichTick of  
    1 -> tick_1(0);  
    2 -> tick_2(0);  
    3 -> tick_3(0)  
  end.
```

```
tick_1(Iter) ->  
  io:format("[~p] Tick: ~p~n", [node(), Iter]),  
  wait(1000),  
  tick_1(Iter+1).
```

...Linked Processes...

```
tick_2(Iter) ->  
  io:format("[~p] Tick: ~p~n", [node(), Iter]).
```

```
tick_3(Iter) ->  
  io:format("[~p] Tick: ~p~n", [node(), Iter]),  
  throw("some exception").
```

```
wait(Delay) ->  
  receive  
    after Delay -> true  
  end.
```

...Linked Processes...

```
loop(WhichTick) ->
  receive
    {'EXIT', Pid, Why} ->
      io:format("Process ~p exited due to ~p -
                restarting locally~n", [Pid, Why]),
      try select_tick(WhichTick) of
        ok -> { normal_exit } % ok from io:format
      catch
        X -> {X} % "some exception"
      end
    end.
end.
```

...Linked Processes...

- Node bob – execute `local(['alice@cko3','1'])` to spawn process to run on remote node alice (no interactive shell):

```
erl -sname bob -setcookie secretcookie  
    -noshell -run link local alice@cko3 1
```

- Node alice (with shell):

```
erl -sname alice -setcookie secretcookie
```


...Linked Processes

- Demo

Dynamic Code Loading...

- Can have two copies of a module running at one time – current version and old version
- Recompiling a module causes:
 - code running in old version to be killed
 - current version becomes old version
 - functions running in loops continue with new current version

...Dynamic Code Loading...

```
-module(update1).  
-export([start/0, tick/2, call_update/1, update/1]).  
-vsn(1.0).  
  
% Register process executing tick() as tickproc  
start() ->  
    register(tickproc,  
        spawn(?MODULE, tick, [?MODULE, 0])), %% MFA  
    wait(100000).
```

...Dynamic Code Loading...

```
% tick, waiting for update message
tick(Module, Iter) ->
  io:format("[~p] Tick: ~p~n", [node(), Iter]),
  receive
    { update, NewModule } ->
      io:format("Updating...~n"),
      NewModule:tick(NewModule, Iter+1)
  after 1000 -> true
end,
Module:tick(Module, Iter+1).

% send update message
update(NewModule) -> tickproc ! {update, NewModule}.
```

...Dynamic Code Loading...

```
% call OldModule:update(NewModule)
call_update([Host, OldModule, NewModule]) ->
    rpc:call(list_to_atom(Host),
    list_to_atom(OldModule), update,
    [list_to_atom(NewModule)]),
init:stop().

wait(Delay) ->
    receive
        after Delay -> true
    end.
```

...Dynamic Code Loading...

```
% module update2 is the same as update1,  
% except the message printed is different
```

```
-module(update2).
```

```
...
```

```
tick(Module, Iter) ->  
    io:format("[~p] Updated tick: ~p~n",  
        [node(), Iter]),
```

```
...
```

...Dynamic Code Loading...

- Node bob – execute `update1:start()`

```
erl -sname bob -setcookie secretcookie  
-noshell -run update1 start
```

- Node alice - execute either:

```
erl -sname alice -setcookie secretcookie -noshell  
-run update1 call_update bob@cko3 update1 update2
```

```
erl -sname alice -setcookie secretcookie -noshell  
-run update2 call_update bob@cko3 update2 update1
```

...Dynamic Code Loading

- Demo

OTP

- Originally "Open Telecom Platform"
- Applications, e.g.:
 - Orber – CORBA ORB
 - Mnesia – distributed database
 - wxErlang - GUI
- Architectural patterns (behaviors), e.g.:
 - gen_server – server side of client/server app
 - gen_fsm – finite state machine
 - gen_event – event handling

Interoperability

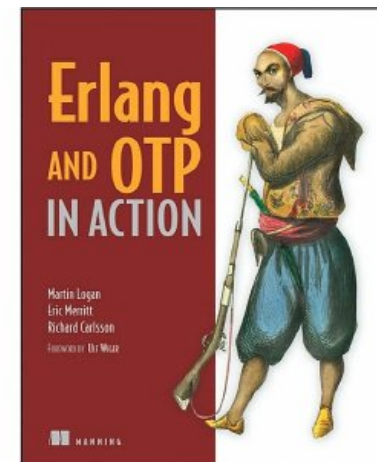
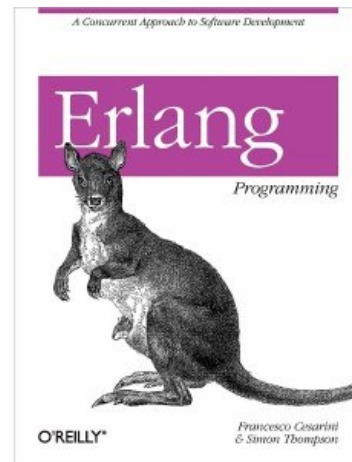
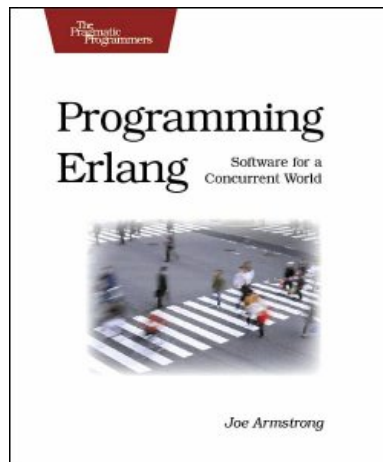
- Port
 - C/C++ process spawned by Erlang
 - Process interprets a byte stream
- C Node
 - Independent C/C++ process
 - not spawned by Erlang
 - Process registers itself as an Erlang node
 - Receives and sends Erlang messages
- Native Implemented Function (NIF)
 - C function in shared library loaded into VM

References

Armstrong. *Programming Erlang, Software for a Concurrent World*. Pragmatic Bookshelf, 2007.

Cesarini, Thompson. *Erlang Programming*. O'Reilly, 2009.

Logan, Merritt, Carlsson. *Erlang and OTP in Action*. Manning, 2011.



References

- www.trapexit.org
<http://www.trapexit.org/>
- Open Source Erlang
<http://www.erlang.org/>
- erldocs.com
<http://erldocs.com/>
- Using Erlang with CORBA and DDS
<http://mnb.ociweb.com/mnb/MiddlewareNewsBrief-201101.html>