Principles of Programming Languages (E)

Matteo Pradella

December 1, 2017

Erlang: Overview

1. Sequential programming

2. Concurrent Programming

Introduction

1. Erlang was introduced in 1986 by Joe Armstrong, Robert Virding, and Mike Williams, working at Ericsson
2. initially born for telecommunication applications (switches and similar stuff)
3. **concurrent-oriented** programming language, **distribution** is almost transparent
4. its core is **functional**; not pure like Haskell, but more pragmatic, as suits its industrial setting
5. syntax heavily influenced by its original **Prolog** implementation
6. **dynamic typed** like Scheme
7. solid standard library for distributed fault-tolerant applications, called **OTP** (Open Telecom Platform); support **continuously running** applications and updates with **code swap**

The Erlang VM (BEAM)

1. Erlang programs run on an ad hoc VM, called BEAM
2. BEAM is very robust and offers many useful features for parallel and distributed systems, e.g. performance degradation is usually slow, fault-tolerance
3. for these reasons, there are other languages, besides Erlang, that run on it (analogously to the JVM, but of course in a smaller scale), mainly:
4. *Elixir*, a syntactic re-thinking of Erlang (Ruby-inspired), with macros and protocols
5. *LFE* (Lisp Flavoured Erlang), the name says it all

Erlang: usage and relevance

1. Erlang is not really "mainstream" nowadays, still it is used in some relevant industrial applications, such as: Whatsapp, Call of Duty (servers), Amazon (SimpleDB), Yahoo! (Delicious), Facebook (Chat), Pinterest (actually uses Elixir).
2. But its most important aspect for us is its present conceptual relevance: new languages and frameworks borrow much from Erlang, consider e.g. Akka (Scala/Java), the so-called "Reactive Manifesto"...
3. Main points: robust distributed computing is more relevant than ever; also consider that new processor architectures can be considered as like miniature distributed systems!

Syntax: Variables

1. Variables start with an Upper Case Letter (like in Prolog).
2. Variables can only be bound **once**! The value of a variable can never be changed once it has been set.

```
Abc
A_long_variable_name
ACamelCaseVariableName
```

Atoms

1. Atoms are like **symbols** in Scheme.
2. Any character code is allowed within an atom, singly quoted sequences of characters are atoms (not strings).
3. unquoted must be **lowercase**, to avoid clashes with **variables**

```
abcef
start_with_a_lower_case_letter
'Blanks can be quoted'
'Anything inside quotes \n'
```

Tuples

1. Tuples are used to store a fixed number of items.

```
{123, bcd}
{123, def, abc}
{person, 'Jim', 'Austrian'}  % three atoms!
{abc, {def, 123}, jkl}
```

1. There is also the concept of **record** (a.k.a. struct), but in Erlang it is just special syntax for tuples.

Lists

1. Are like in Haskell, e.g. `[1, 2, 3]`, ++ concatenates
2. main difference: `[X | L]` is cons (like `(cons X L)`)
3. Strings are lists, like in Haskell, but is getting common to use *bitstrings* and UTF.
4. Comprehensions are more or less like in Haskell:

   ```
   > [{X,Y} || X <- [-1, 0, 1], Y <- [one, two, three], X >= 0].
   [{0,one},{0,two},{0,three},{1,one},{1,two},{1,three}]
   ```
5. Indeed there is nice syntax and facilities for sequences of bits, also comprehensions

Pattern Matching

1. Like in Prolog, = is for **pattern matching**; _ is "don't care"

```
A = 10
  Succeeds - binds A to 10
{A, A, B} = {abc, abc, foo}
  Succeeds - binds A to abc, B to foo
{A, A, B} = {abc, def, 123}
  Fails
[A,B|C] = [1,2,3,4,5,6,7]
  Succeeds - binds A = 1, B = 2, C = [3,4,5,6,7]
[H|T] = [abc]
  Succeeds - binds H = abc, T = []
{A,_, [B|_], {B}} = {abc, 23, [22, x], {22}}
  Succeeds - binds A = abc, B = 22
```

Maps

1. There are the (relatively new) maps, basically hash tables.
2. Here are some examples:

```
> Map = #{one => 1, "Two" => 2, 3 => three}.
#{3 => three,one => 1,"Two" => 2}
> % update/insert
> Map#{one := "I"}.
#{3 => three,one => "I","Two" => 2}
> Map.
#{3 => three,one => 1,"Two" => 2} % unchanged
> % I want the value for "Two":
> #{"Two" := V} = Map.
#{3 => three,one => 1,"Two" => 2}
> V.
2
```

Function Calls

```
module:func(Arg1, Arg2, ... Argn)
func(Arg1, Arg2, .. Argn)
```

1. Function and module names (func and module in the above) must be atoms.
2. Functions are defined within Modules.
3. Functions must be exported before they can be called from outside the module where they are defined.
4. Use `-import` to avoid qualified names, but it is discouraged

Module System

```
-module(demo).
-export([double/1]).
double(X) ->
  times(X, 2).
times(X, N) ->
  X * N.
```

1. double can be called from outside the module, times is local to the module.
2. double/1 means the function double with one argument (Note that double/1 and double/2 are two different functions).
3. symbols starting with '-' are for the **preprocessor** (analogous to cpp), while macro calls start with '?'

Starting the system

```
shell> erl
...
Eshell V8.2 (abort with ^G)
1> c(demo).
double/1 times/2 module_info/0
compilation_succeeded
2> demo:double(25).
50
3> demo:times(4,3).
** undefined function:demo:times[4,3] **
** exited: {undef,{demo,times,[4,3]}} **
```

There are also `erlc` for compiling, `escript` for running scripts, etc.

Built In Functions (BIFs in Erlang jargon)

1. BIFs are in the `erlang` module
2. They do what you cannot do (or is difficult to do, or too slow) in Erlang, and are usually implemented in C.

```
date()
time()
length([1,2,3,4,5])
size({a,b,c})
atom_to_list(an_atom)      % "an_atom"
list_to_tuple([1,2,3,4])  % {1,2,3,4}
integer_to_list(2234)      % "2234"
tuple_to_list({})  ...
```

Function Syntax & Evaluation

1. A function is defined as a sequence of clauses.

```
func(Pattern1, Pattern2, ...) -> ... ;
func(Pattern1, Pattern2, ...) -> ... ;
...
func(Pattern1, Pattern2, ...) -> ... .
```

2. Clauses are scanned sequentially until a match is found.
3. When a match is found all variables occurring in the head become bound.
4. Variables are local to each clause, and are allocated and deallocated automatically.
5. The body is evaluated sequentially (use "," as separator).

Functions (cont)

```
-module(mathStuff).
-export([factorial/1, area/1]).

factorial(0) -> 1;
factorial(N) -> N * factorial(N-1).

area({square, Side}) ->
  Side * Side;
area({circle, Radius}) ->
  3.14 * Radius * Radius;
area({triangle, A, B, C}) ->
  S = (A + B + C)/2,
  math:sqrt(S*(S-A)*(S-B)*(S-C));
area(Other) ->
  {invalid_object, Other}.
```

Guarded Function Clauses

```
factorial(0) -> 1;
factorial(N) when N > 0 ->
  N * factorial(N - 1).
```

1. The keyword **when** introduces a guard, like | in Haskell.

Examples of Guards

```
number(X)             - X is a number
integer(X)            - X is an integer
float(X)              - X is a float
atom(X)               - X is an atom
tuple(X)              - X is a tuple
list(X)               - X is a list
X > Y + Z             - X is > Y + Z
X =:= Y               - X is exactly equal to Y
X =/= Y               - X is not exactly equal to Y
X == Y                - X is equal to Y
  (with int coerced to floats,
   i.e. 1 == 1.0 succeeds but 1 =:= 1.0 fails)
length(X) =:= 3       - X is a list of length 3
size(X) =:= 2         - X is a tuple of size 2.
```

1. All variables in a guard must be bound.

Apply

```
apply(Mod, Func, Args)
```

1. Apply function Func in module Mod to the arguments in the list Args.
2. Mod and Func must be atoms (or expressions which evaluate to atoms).

   ```
   apply(?MODULE, min_max, [[4,1,7,3,9,10]]).
   {1, 10}
   ```

3. Any Erlang expression can be used in the arguments to apply.
4. `?MODULE` uses the preprocessor to get the current module's name

Other useful special forms

```
case lists:member(a, X) of
  true ->    ... ;
  false ->   ...
end,

if
  integer(X) ->  ... ;
  tuple(X) ->    ... ;
  true ->        ... % works as an "else"
end,
```

Note that `if` needs **guards**, so for user defined predicates it is customary to use `case`.

Lambdas and Higher Order Functions

1. Syntax for lambdas is, e.g., `Square = fun (X) -> X*X  end.`
2. We can use it like this: `Square(3).`
3. Lambdas can be passed as usual to higher order functions:
   `lists:map(Square, [1,2,3]).` returns [1,4,9]
4. To pass standard (i.e. "non-lambda") functions, we need to prefix their name with `fun` and state its arity:
   `lists:foldr(fun my_function/2, 0, [1,2,3]).`

Concurrent Programming: the Actor Model

1. The Actor Model was introduced by Carl Hewitt, Peter Bishop, and Richard Steiger in 1973
2. Everything is an actor: an independent unit of computation
3. Actors are inherently concurrent
4. Actors can only communicate through messages (async communication)
5. Actors can be created dynamically
6. No requirement on the order of received messages

Concurrency oriented programming language

1. Writing concurrent programs is easy and efficient in Erlang
2. Concurrency can be taken into account at early stages of development
3. Processes are represented using different actors communicating only through messages
4. Each actor is a lightweight process, handled by the VM: it is not mapped directly to a thread or a system process, and the VM schedules its execution
5. The VM handles multiple cores and the distribution of actors in a network
6. Creating a process is fast, and highly concurrent applications can be faster than the equivalent in other programming languages

Concurrent programming

There are three main primitives:

1. `spawn` creates a new process executing the specified function, returning an identifier
2. `send` (written `!`) sends a message to a process through its identifier; the content of the message is simply a variable. The operation is asynchronous
3. `receive ... end` extract, going from the first, a message from a process's mailbox queue matching with the provided set of patterns – this is blocking if no message is in the mailbox. The mailbox is persistent until the process quits.

Creating a New Process

1. we have a process with `Pid1` (**Process Identity** or Pid)
2. in it we perform Pid2 = spawn(Mod, Func, Args)
3. like `apply` but spawning a new process
4. after, `Pid2` is the process identifier of the new process – this is known only to process Pid1.

Simple Message Passing

1. Process A sends a message to B (it uses self() to identify itself)
   Pid_B `! {self(), foo}`
2. $\{Pid_A$, foo$\}$ is sent to process B
3. B receives it with

```
receive
   {From, Msg} -> Actions
end
```

4. self() – returns the Pid of the process executing it
5. From and Msg become bound when the message is received.

Simple Message Passing (2)

1. Process *A* performs
   Pid_B `! {self(), {mymessage, [1,2,3]}}`
2. *B* receives it with

```
receive
   {A, {mymessage, D}} -> work_on_data(D);
end
```

3. Messages can carry data and be selectively unpacked
4. Variables A and D become bound when receiving the message
5. If A is bound before receiving a message, then only data from that process is accepted.

An Echo process (1)

```
-module(echo).
-export([go/0, loop/0]).

go() ->
  Pid2 = spawn(echo, loop, []),
  Pid2 ! {self(), hello},
  receive
    {Pid2, Msg} ->
      io:format("P1 ~w~n",[Msg])
  end,
  Pid2 ! stop.
```

An Echo process (2)

```
loop() ->
  receive
    {From, Msg} ->
      From ! {self(), Msg},
      loop();
    stop ->
      true
  end.
```

Selective Message Reception

1. A performs Pid_C ! foo
2. B performs Pid_C ! bar
3. code in C:

```
receive
   foo -> true
end,
receive
   bar -> true
end
```

4. `foo` is received, then `bar`, irrespective of the order in which they were sent.

Selection of any message

1. A performs Pid_C ! foo
2. B performs Pid_C ! bar
3. code in C:

```
receive
   Msg -> ... ;
end
```

4. The first message to arrive at the process C will be processed – the variable Msg in the process C will be bound to one of the atoms foo or bar depending on which arrives first.

Registered Processes

1. register(Alias, Pid) Registers the process Pid with name Alias

```
start() ->
  Pid = spawn(?MODULE, server, [])
  register(analyzer, Pid).

analyze(Seq) ->
  analyzer ! {self(), {analyze, Seq}},
  receive
    {analysis_result, R} -> R
  end.
```

2. Any process can send a message to a registered process.

Client Server Model (1)

1. Client-Server can be easily realized through a simple protocol, where requests have the syntax `{request, ...}`, while replies are written as `{reply, ...}`
2. Server code

```
-module(myserver).
server(Data) ->  % note: local data
  receive
  {From,{request,X}} ->
    {R, Data1} = fn(X, Data),
    From ! {myserver,{reply, R}},
    server(Data1)
  end.
```

Client Server Model (2)

1. Interface Library

```
-export([request/1]).
request(Req) ->
  myserver ! {self(),{request,Req}},
  receive
    {myserver,{reply,Rep}} -> Rep
  end.
```

Timeouts

1. consider this code in process *B*:

```
receive
  foo   -> Actions1;
after
  Time -> Actions2;
```

2. If the message foo is received from *A* within the time Time perform Actions1 otherwise perform Actions2.

Uses of Timeouts (1)

1. sleep(T) – process suspends for T ms.

```
sleep(T) ->
  receive
  after
    T -> true
  end.
```

2. suspend() – process suspends indefinitely.

```
suspend() ->
  receive
  after
    infinity -> true
  end.
```

Uses of Timeouts (2)

1. The message What is sent to the current process in T ms from now

```
set_alarm(T, What) ->
  spawn(timer, set, [self(), T, What]).

set(Pid, T, Alarm) ->
  receive
  after
    T -> Pid ! Alarm
  end.
receive
  Msg -> ... ;
end
```

Uses of Timeouts (3)

1. flush() – flushes the message buffer

```
flush() ->
  receive
    Any -> flush()
  after
    0 -> true
  end.
```

2. A value of 0 in the timeout means check the message buffer first and if it is empty execute the following code.

Building reliable and scalable applications with Erlang

1. **OTP** (Open Telecom Platform): set of libraries and of design principles for Erlang industrial applications
2. **Behaviours** (note the British spelling): ready-to-use design patterns (Server, Supervisor, Event manager . . .), only the functional part of the design has to be implemented (callback functions)
3. Applications structure, with supervision; **"let it crash"** principle
4. Support to code **hot-swap**: application code can be loaded at runtime, and code can be upgraded: the processes running the previous version continue to execute, while any new invocation will execute the new code

"Let it crash": an example

1. We are going to see a simple supervisor **linked** to a number of workers.
2. Each worker has a state (a natural number, 0 at start), can receive messages with a number to add to it from the supervisor, and sends back its current state. When its local value exceeds 30, a worker ends its activity.
3. The supervisor sends "add" messages to workers, and keeps track of how many of them are still active; when the last one ends, it terminates.
4. We are going to add code to simulate random errors in workers: the supervisor must keep track of such problems and re-start a new worker if one is prematurely terminated.

Code: the main function

```
main(Count) ->
    register(the_master, self()), % I'm the master, now
    start_master(Count),
    unregister(the_master),
    io:format("That's all.~n").
```

Code: starting the master and its children

When two process are linked, when one dies or terminates, the other is killed, too. To transform this kill message to an actual manageable message, we need to set its `trap_exit` process flag.

```
start_master(Count) ->
    % The master needs to trap exits:
    process_flag(trap_exit, true),
    create_children(Count),
    master_loop(Count).

% This creates the linked children
create_children(0) -> ok;
create_children(N) ->
    Child = spawn_link(?MODULE, child, [0]),  % spawn + link
    io:format("Child ~p created~n", [Child]),
    Child ! {add, 0},
    create_children(N-1).
```

Code: the Master's loop

```
master_loop(Count) ->
    receive
        {value, Child, V} ->
            io:format("child ~p has value ~p ~n", [Child, V]),
            Child ! {add, rand:uniform(10)},
            master_loop(Count);
        {'EXIT', Child, normal} ->
            io:format("child ~p has ended ~n", [Child]),
            if
                Count =:= 1 -> ok; % this was the last
                true -> master_loop(Count-1)
            end;
        {'EXIT', Child, _} ->  % "unnormal" termination
            NewChild = spawn_link(?MODULE, child, [0]),
            io:format("child ~p has died, now replaced by ~p ~n",
                      [Child, NewChild]),
            NewChild ! {add, rand:uniform(10)},
            master_loop(Count)
    end.
```

Code: Children's main loop

```
child(Data) ->
    receive
        {add, V} ->
            NewData = Data+V,
            BadChance = rand:uniform(10) < 2,
            if
                % random error in child:
                BadChance -> error("I'm dying");
                % child ends naturally:
                NewData > 30 -> ok;
                % there is still work to do:
                true -> the_master ! {value, self(), NewData},
                        child(NewData)
            end
    end.
```

Let's run it

```
1> exlink:main(3).
Child <0.68.0> created
Child <0.69.0> created
Child <0.70.0> created
child <0.68.0> has value 0
child <0.69.0> has value 0
child <0.70.0> has value 0
child <0.68.0> has value 5
child <0.69.0> has value 2
child <0.70.0> has value 6
child <0.68.0> has value 12
child <0.69.0> has value 12
child <0.70.0> has value 16
child <0.68.0> has value 16
```

run (cont.)

```
child <0.69.0> has value 15
child <0.70.0> has value 22
child <0.68.0> has value 23
=ERROR REPORT==== 9-Jan-2017::12:17:46 ===
Error in process <0.70.0> with exit value:
{"I'm dying",[{exlink,child,1,[{file,"exlink.erl"},{line,17}]}]}
child <0.69.0> has value 19
child <0.70.0> has died, now replaced by <0.71.0>
child <0.68.0> has value 27
child <0.69.0> has value 25
child <0.71.0> has value 9
child <0.68.0> has value 29
child <0.69.0> has ended
child <0.71.0> has value 13
child <0.68.0> has ended
```

run (cont.)

```
child <0.71.0> has died, now replaced by <0.72.0>
=ERROR REPORT==== 9-Jan-2017::12:17:46 ===
Error in process <0.71.0> with exit value:
{"I'm dying",[{exlink,child,1,[{file,"exlink.erl"},{line,17}]}]}
child <0.72.0> has value 2
child <0.72.0> has value 9
child <0.72.0> has value 13
child <0.72.0> has value 22
child <0.72.0> has value 27
child <0.72.0> has ended
That's all.
ok
```

Legal stuff

Acknowledgments: Some examples and inspiration are from the old official Erlang tutorial and Alessandro Sivieri's old slides for this same course.