Principles of Programming Languages: Functional Erlang

Alessandro Sivieri, PhD

A.Y. 2013/2014

Outline

Why Erlang?

The world is parallel.

If you want to write programs that behave as other objects behave in the real world, then these programs will have a concurrent structure.

Use a language that was designed for writing concurrent applications, and development becomes a lot easier.

Erlang programs model how we think and interact.

Erlang history

- 1982-1986. Programming experiments: how to program a telephone exchange
- 1986. Erlang emerges as a dialect of Prolog. Implementation is a Prolog interpreter. Author: Joe Armstrong
- 1986. New abstract machine, called JAM
- 1993. Turbo Erlang (BEAM compiler)
- 1993. Distributed Erlang
- 1996. OTP (Open Telecom Platform)
- 1996. AXD301 switch announced: over a million lines of Erlang, reliability of nine 9s

Erlang history

- 1998. Erlang banned within Ericsson for other products
- 1998. Erlang fathers quit Ericsson
- 1998. Open source Erlang
- 2004. Armstrong re-hired by Ericsson
- 2006. Native symmetric multiprocessing is added to runtime system
- 2014. April. Latest stable release: 17.0

Erlang today

- Ericsson, Amazon, Yahoo!, Huffington Post...
- CouchDB
- RabbitMQ
- Github
- Facebook
- WhatsApp
- Online gaming services (e.g., Call of Duty)

Erlang

Erlang is a *functional* and *concurrent* programming language

Functional

- Functions are first-class values
- Computation is performed through mathematical function evaluation

Concurrent

- Actor model
- Asynchronous message passing
- Efficient concurrency management

Erlang

- Erlang runs on the BEAM emulator, or can be compiled to native code (HiPE)
- A program is compiled to bytecode and then executed inside the emulator
- It offers an interactive shell

Outline

Variables

- Variables must start with a capital letter
- Variables are untyped
  - A = 123456789
  - B = “erlang”
  - C = 123.456 * 789.012
- Single assignment
- Types: integer (arbitrary precision), float (arbitrary precision), list, tuple, bitstring, lambda function (*fun*), atoms

Atoms

- Atoms represent non-numerical constant values
- Atoms start with a lowercase letter
- Atoms are global
- Its value is the atom itself
- If atoms contain some characters (e.g., “-”), you may need to use ticks to delimit them
- Function names, module names, host names are all atoms

Tuples

- A tuple is composed by a fixed number of unnamed fields (e.g., `{temp, 12}`)
- Since there is no concept of class or struct, a tuple becomes central in holding a set of related values
- A best practice is to set the first element of a tuple to be an atom describing the *type* the tuple is an instance of
- Erlang offers some syntactic sugar to enhance this behavior, with the so-called *records*

Pattern matching

- Each line in Erlang is actually a pattern match
- The interpreter evaluates the right-hand side of "=" and tries to match the result against the left-hand side
- If a match fails, then an error code is generated
- This is used to extract values from tuples

```
Point = {point, 10, 12},
{point, X, Y} = Point,
{_, _, W} = Point.
```

Lists

- Lists can have heterogeneous elements: `[erlang, 10, {lemon, 3}]`
- We can pattern match head and tail
- Erlang supports list comprehensions
- Strings are lists of (ASCII) *integers*

```
Buy = [{apple, 10}, {pear, 12}, {orange, 4}, {lemon, 6}],
[AppleTuple, PearTuple|Others] = Buy,
NewBuy = [{milk, 2}|Buy],
NewList = [X || X <- [1, 2, a, 3, 4, b, 5, 6], integer(X), X > 3].
```

Other types

- Dictionary (module *dict*): performs as a 2D tuple, where each first element is the key and the second is the value
- Constant values: defined using the `define` keyword, recalled in code prepending a question mark to the name (usually the name is all uppercase)
- There are some other types: pid (see the *actors* section), port, reference, map (this one is experimental in R17.0)

Functions

- A function is univocally identified by name and arity
- Each function can have multiple clauses, and pattern match on the structure of the passed variables defines the clause to be executed
- If the arity is different, they are different functions

```
area({square, X}) -> X * X;
area({rect, X, Y}) -> X * Y;
area({circle, R}) -> 3.14 * R.

area(X) -> X * X.

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

Functions

- We can reason on variable values using *guards*
- Multiple conditions can be combined (*and, or, andalso, orelse*)
- Only a *small* subset of functions can be used in guards (*no* user-defined functions at all)

```
max(X, Y) when X > Y -> X;
max(X, Y) -> Y.
```

Functions

- We can assign functions to variables
- We can write lambda functions, and use higher-order functions (e.g., *map, filter, fold...*)

```
D = fun(X) -> X * 2 end,
A = [1, 2, 3],
B = lists:map(D, A).
```

Control structures

- The `case` expression pattern matches on a variable
- If no pattern matches, the code *fails*
- The `if` expression is used *only* with guard conditions

```
case Expression of
  Pattern1 [when Guard1] -> Expr_1;
  Pattern2 [when Guard2] -> Expr_2;
  ...
  Any -> io:format("Unknown pattern: ~p~n", [Any])
end.

if
  Guard1 -> Expr_1;
  Guard2 -> Expr_2;
  ...
  true -> Default
end.
```

Control structures

- Since Erlang is a functional language, there is no concept of *while* or *for* loops…
- …but, since Erlang is a functional language, we can use recursion and assign functions to variables
- The interpreter *does* perform tail recursion in constant stack space, so you need to use it (or use library functions)

Extras

- Erlang supports bit sequences: pattern matching can be performed on a sequence of bits (specifying number of bits, endianess, signedness)
- Erlang supports exception handling

```
try Expression of
  Pattern1 [when Guard1] -> Expr_1;
  Pattern2 [when Guard2] -> Expr_2;
  ...
catch
  ExceptionType: ExPattern1 [when Guard1] -> Expr_1;
  ExceptionType: ExPattern2 [when Guard2] -> Expr_2;
  ...
after
  AfterExpression
end.
```