Note slide Distributed Systems 3

Erlang: Simple Language Suited to...

Erlang is a quite simple language.
"Simple" means that it is not a language with so many different rules, with a complex structure and so on.

C++ is a very complex and difficult language.

Erlang is suitable to develop concurrent applications with a fine-grained parallelism, so small parallel components that interact across them.

It is used in network servers.

It is also suited to develop machinery for middleware:
it has been used in distributed dbs, and to develop servers to support the distribution of asynchronous messages (message queue servers);

and it is suited to develop soft real-time apps for the purposes of monitoring, controlling other apps, some devices, testing tools and so on.

It is not a language for large scale apps. For this kind of apps, a new language has been proposed, named Elixir.

Then, why we are talking about Erlang?

Because it has, for the purposes of the course, really interesting features.

It adopts message passing:
you can put your hands directly on making different components, interacting via message passing without taking care of all the underlying network infrastructure, because the communication primitives are available to you directly at the language level, so this is extremely useful for us.

It is also important for us to become familiar with a functional programming paradigm.

System Overview (I)

Overview of the overall Erlang system.

The Erlang code is executed by the so called ERTS (Erlang Run-Time System).
As many other languages of this kind, it is executed by means of an intermediate language, bytecode, so we need to compile Erlang source code.
Once the code has been compiled, it can be first loaded and then run over a special dedicated virtual machine, BEAM.

Erlang programs are organized in separate files containing source code and their extension is .erl.

All the SW is organized in basic application blocks, named modules.
One module at a time is compiled and if you have one module in an erl file, once it is compiled, a corresponding .beam file will be generated.
The VM (BEAM) will load the bytecode in that .beam file and it will execute such code.

One important fact is that the BEAM is just one single process at the OS level --> all concurrency issues are dealt with just inside that OS process, so BEAM does not rely on threads f.i. provided by the OS, it handles everything just internally.
So, as we enter the BEAM door, inside it everything is organized according to the Erlang rules, it is a new world, we do not have to imagine how to map concurrence units inside the BEAM and resources at the OS level.

Anyway, BEAM is typically run over multicore CPUs, so it has the possibility to exploit … CPUs, and internally it has a number of schedulers to distribute the execution of different concurrent units, Erlang programs, over the available cores.
So, the fact that it is just one single process doesn't hamper (ostacolare) the exploitation of the underlying cores.

Remember:
Erlang processes do not correspond to any kind of processes or threads at the OS level.
This is important because BEAM has been organized to provide a very lightweight support to Erlang processes, and typically this support is much more lightweight than those provided for threads in OSs.

System Overview (II)

We have to deal with dynamic memory allocation, and, as in many other languages, in ERTS we have a garbage collector, or at least we have a garbage collection system.

How to interact with the system ?
There is the possibility to make use of a shell, named usually REPL (Read, Execute, Print, Loop).
So you can write your commands and statements in the shell and they will be executed line by line, chunk by chunk.

How to start the shell?
You have to type erl.

An expression is terminated with a period "." --> "(2+3.)".

Executed means that the single expression will be evaluated.

There is a counter for each line at the beginning of the line.

How to compile and load pieces of SW ?
SW has to be written inside special files. If I want to compile one single module in the

corresponding file, you can do it directly from the shell.
You can imagine Erlang modules exactly like Java classes.

In the shell we can evaluate expressions but there is no possibility for defining functions. Functions are defined inside modules, and functions are the most important part of the language.
But, if there is a function available from the shell, maybe in an already compiled module, you can call functions within the shell because calling functions is just an operation that corresponds to the evaluation of an expression.
So, functions exported by modules can be accessed after module loading.

System Overview (III)

Erlang apps are independent of both the HW and the OS. This is important because it is a crucial feature for distributed apps.
F.i. Java has become the primary language for distributed apps just because the use of the VM enabled different pieces of app to be spread across heterogeneous machines.
Here we have the same situation.
As we will see, Erlang processes can be spawn on different nodes, and the communication across processes both within the same node and across nodes is basically the same, can be specified in the same way.
So, the placement of processes on different nodes does not affect, in some extent, the way you have to develop your program.

One of the reasons of the success of Java is in the fact that it has a very reach standard library and Java programmers can rely on functionalities already coded in the standard library (API), and this library is automatically loaded by any Java installation.

In Erlang there is the support of a sort of library and other components and different templates, behaviors and so on and it will act as a crucial helper for the Erlang programmer. Somehow it can be viewed as a sort of middleware for Erlang apps. You can see in the picture:
At the lower level you have the HW and the OS;
over them you have the ERTS;
over it you have the Erlang apps.
You can notice that Erlang apps are completely separated by the HW and OS, but then they can take advantage of standard libraries and all the components of the OTP (Open Telecom Platform):
it was originally developed for telecommunication purposes, to support the development of SW to be deployed on network nodes, later is has become a general middleware tool.

Predictable Prelude: Expressions etc..

Now we will talk about what expressions are, how they are evaluated, composed and so on.

An Erlang program is made of expressions, and a program is run by evaluating such expressions. They are listed one after the other and the execution of the program corresponds to the evaluation of such expressions.

How are expressions structured ?

The basic component of an expression is named term:

a piece of data of any of the data types supported by the language. It can be written used the corresponding literals.
A literal corresponds to the rule of writing the term using the ordinary character set.
The term evaluates (in jargon “returns”) to the term itself.
If I write 1 the evaluation of 1 is just 1, if I write “people” the evaluation is “people” itself.

Expressions can be made of sub expressions which are combined using operators.
All the sub-expression has to be evaluated before the expression itself is evaluated.

Variable: it is itself an expression.
If a variable is bound to a certain value, the evaluation of a variable corresponds to such a value (if x is bound to 42, the evaluation of x corresponds to the value 42)

Starting Up: Expressions, Numbers

We start with the simplest terms.

In the shell an expression must be terminated by “.”.
Multiple subsequent expressions must be separated by the “,”.
If the expression is made of several separated expression by a “,”, the evaluation of the whole command corresponds to the value returned by the evaluation of the last term.

Considering numbers, we have 2 types of numeric literals:

- Integers
- Floats

You can use this literal, base#value, to express an int value in any base. F.i. 8#4712 = 2506(in decimal)

Another useful literal is this $char:
it will return the ASCII code for that character. F.i. $r = 114

There is an operator called rem that stands for reminder.

Variables (so to speak)...

Now we will talk about variables.

They are named variables, but they are not allowed to change their value --> SINGLE ASSIGNEMENT.

Variables must start with a capital letter, or with the underscore.

What about the type of a variable ?

Erlang is dynamically typed.

Variables have to be bound, but how to do it ?

By applying pattern matching:
it is maybe the most important concept in Erlang.

What is a pattern?

A pattern is structured like a term but it may include unbounded variables.
If I write “X.” this is an unbounded variable so it is a pattern.

There is a special variable named anonymous variable and its name is “_”. It can be used when a variable is required but its value can be ignored. In this case I can try to assign subsequently many values to this variable, pattern matching operation will succeed but I will not make use of such values in the program.

Pivotal Concept: Pattern Matching

The main idea of pattern matching is to have a pattern on the left side and a term on the right side.
These 2 parts are compared using the pattern matching operator which corresponds in the most trivial case to the equals character (“=”).

We try to make both sides identical by binding unbounded variables of the pattern to special values and in case this will be possible, we will have an actual match, otherwise there will be an error.

In this case you can see the ordinary assignment operators of other languages as a sort of special case of this pattern matching operation.

Line 2: A = 2 --> in case A is an unbounded variable this will succeed only by binding A to the value 2. After this binding, you have 2 on one side and 2 on the other side, that’s it. But this can be generalized in a different way.

Line 3: on the left you have something that contains variables, on the other side you have something that contains values. How could you make the 2 parts matching ?
Only by making A binding to 0 and B binding to 1. But here there is a problem: variable A is already bound to 2, so this turn to be “{2, (B=)0, 1}”. There is no way to make these 2 parts identical, so this turns to be an error condition.

When pattern matching is used?

Often in the evaluation of a function call and with other types of expressions like case, receive, try.
In case the matching fails, we face a runtime error.

Basic Data Types

Even though we do not necessarily specify the type of our variables, types are present, and there are some basic data types.

In this slide you can see a list of them.

- There is no Boolean but some special atoms (true and false) are used.
- There are funs: in other languages they are called lambda.
- There are Pids used to identify a process.

Beside these ones there are also compound data types:

- Tuples: a collection of a fixed number of terms that cannot be modified.
- Record: it is just another way to access to tuple using named fields.
- List: the first or second important one. It is thought to deal with a variable number of elements.
- Strings: they are not actually a separate data type; they correspond to a list of integers.
- Maps: like dictionary in Python, with variable size.

atoms

Let's start from ATOMS.

It is a special data type introduced to support to represent a constant value, in a way that could be meaningful to the programmer.

They start with lowercase letter.
So, an atom value is unique within an entire program.
In some sense they are similar to the enumerated types in C and Java.
See in the picture, "lion" does not start with a capital letter, so it is not a variable but an atom.

The evaluation of the atom corresponds to the atom itself.
As it is a data type we can do like at expression 2:
I'm trying to apply pattern matching; on the left we have a variable (Animal), the result of this pattern matching op is just the value for both sides after the association of the variable with the corresponding value (lion), so the result of the evaluation is lion itself.

{ Tuples }

How grouping together terms or elements?
Through tuples.

We can group he elements by using special delimiters "{ }".

It has a fixed number of terms or elements.

They resemble structure in C, but here we do not have named fields.

We can use them in a way that the values in them can be extracted and assigned to variables using pattern matching.

Look at the example:
at line 1 we define a tuple with 2 atoms (alan and turing).

I can use a tuple as the value for a variable.
At line 2 we bound a tuple to a variable.

At line 3 I want to assign a special variable named Name to the first element of the tuple.
Suppose we are supposed in matching only the last name:
instead of writing everything exactly with the same value, like at line 3, I can use the anonymous variable “_” like at line 5.

There exist a variant of tuples named record in which you can assign names to the different fields of the tuple.

[Lists]

They are limited by “[ ]”, as in Python.
But we deal with them in a slightly different way.

We want to underline the structure of the list:

- A head: the first element
- A tail: all the other elements --> it is a list itself.

This is a recursive definition of list.

The Empty list is: [ ], that contains nothing.

A list which is not empty, is made of an head (an element) and a tail (a list) and it can be indicated as:
[H | T] (“I” is called “cons”).

Lists can be used in pattern matching:
f.i. pattern matching can be used to assign a value containing a list to a variable, or extracting a sub list and assign it to a variable and so on.

See the picture:

At line 1, we define a list L1.
At line 2, we define yet another list, starting from another one in the tail of the new one (that is a list, so I use L1).
At line 3, we have to assign pattern matching of course: on the left side I have a list made of the Head (which contains an unbounded variable X) and of the Tail (which contains an unbounded variable Y), on the right we have a list L1 --> we will assign X to the Head of L1 and Y to the tail of L1.

Built-in Functions (BIFs)

There exist these system functions, named BIF = Built-In Functions.

They are special functions that have been written directly in a beam file in the VM because developing the corresponding functionalities in Erlang is somehow difficult, not efficient and so on.

Some of them are imported automatically at the beginning, so you can directly use them, as they are, others belong to special part of the standard library and to be executed you have also to specify the module where the function has been defined. F.i. you can see at line 7 function format() (a sort of print op) you have to specify also the module name io.
It is not the same for the function length().
Line 3: define a tuple and refer to it by means of a variable Tup. What if I want to extract the second element from Tup ?
I can use the BIF named element() (line 4).

Line 5: what if I want to access the content in the form of a list ?
may I somehow obtain with exactly the same element, maybe because I want to make some operation over them (and I cannot use tuples for these because they are dedicated only to frozen collection of terms)? There is the tuple_to_list() BIF.

List Comprehensions

They are typical tools of functional languages and Python got this concept from functional languages.
It is a compact notation for generating elements in a list, according to the rules that are specified.
Look at the general syntax: it looks like a list, but inside you find a sort of coding of what to do to obtain all the elements. So you have first have an expression on the left, and then on the right of the “||” delimiter you have several qualifiers that specify what values to consider.
Such values will be then used in the expression to build up each single element.

We have now to specify what a qualifier is:
it can be a generator, a filter, something else..

What about generators?

They are indicated in this way: Pattern <:-- ListExpr.

What about filters?

They are expressions that evaluate to true/false, and so they decide whether that particular element has to be actually taken to be processed or not.

Look at the example:

Line 1: we define a list

Line 2: we build another list through list comprehension, according to a certain rules.
The Expression is X*X, but what is X ?*
X can be extracted one by one by list L1, and this can be expressed using a generator as qualifier:
X (is the Pattern) <:-- L1 (is the list expression).
*So at the beginning we have that X is 1 --> X*X = 1, then X is 2 –> X*X = 4 and so on.

Line 3: The Expression is X.
How to obtain this X ?
X has to be taken from L1, but now there is another rule: the remainder of X divided by 2 doesn't have to be equal to 0 (take all elements in L1 that are not even numbers). In this case we used a generator as before, plus a filter (X rem 2 =/= 0).

Line 4: we build list L2

Line 5: I want to build up a new list according to these rules: a list of tuples (Expression is {X, Y}) and X can be obtained by L2, and Y can be obtained by L2. The result will contain all the possible way to have the pair getting the first element from L2, and the second element from L2 (so the cartesian product between L2 and L2 itself).