Functional Programming in Erlang

Christian Rinderknecht

31 October 2008

Why Erlang?

This course should be devoted to the concepts that are widely implemented in the programming languages available nowadays. So why Erlang only?

The reason is that you already must follow classes in Java and C++, so the **object-oriented programming** is something you are learning anyway.

You also learn C, as a part of C++ or by itself (for system programming), so you are also familiar with **structured programming**.

You also have the possibility to choose my class of *Artificial Intelligence*, which is mainly an introduction to the Prolog programming language, allowing you to gain some understanding of **logic programming**.

Why Erlang? (cont)

What is missing is

- concurrent programming,
- distributed programming,
- functional programming.

Concurrent programming consists in making a software run on separate *threads* or *processes*, i.e. each part of the software is logically running in parallel on the same run-time environment (e.g. a virtual machine or the operating system in case of native-code compilation), therefore on the same operating system (no networking involved).

Why Erlang? (cont)

Of course, if the hardware includes only one processor (CPU), the concurrent execution will not be truly in parallel, but interleaved, i.e. one thread is run for a while, then another is scheduled to use the CPU etc. If several processors are present, true concurrency can be achieved, depending on the compilers.

In the class of *System Programming*, you learn how to program concurrently in C, but the C language, despite it was designed for this task, has no built-in features to handle the concurrency: it relies instead on the operating system, which provides the support for threading and processing.

Because of this dependence by design on low-level layers (i.e. the operating system), which obscures the concepts at stakes, concurrency in C is not an easy task.

Why Erlang? (cont)

Distributed programming is an extension of concurrent programming.

In the former case, the threads are running on different run-time environments on different physical machines, which are interconnected by means of a network (each machine may run several threads of the whole application).

Again, using C for this task, for example, is possible but can be made easier if the programming language includes features for distributing the computations across a network.

Why Erlang? (cont)

Functional programming is a form of *declarative programming* (which includes also logic programming), in which disciple the programmer describes what he wants instead of how to make it.

For example, the language XSLT, which allows the specification of transforms of XML files, is declarative. The query language SQL is also declarative.

Declarative languages are recognisable by the purposeful lack of variables whose values can be modified, in other words: the programmer has no access to the memory whatsoever. In particular, there are no looping constructs, no pointer arithmetics, no explicit dynamic memory allocation or deallocation etc.

The declarative paradigm makes debugging of concurrent and distributed application easier since there are no **side-effects**.

Why Erlang? (cont)

The sequential subset of Erlang is functional because the lack of side-effects eases the debugging.

Concurrency and distribution make the application much more difficult to debug in presence of side-effects and shared memory.

Even in the case of sequential programming (i.e. one thread), if there is no global state shared by several functions and if every function call returns the same result with the same arguments — which is not the case always if side-effects are allowed —, the debugging is much easier.

Introduction

Erlang was developed at Ericsson. Since 1998, it is distributed independently with an open source license.

The main web site distributing Erlang is

`http://www.erlang.org`

The implementation is called **Erlang/OTP** (Open Telecommunication Platform).

Introduction (cont)

Sequential programming in Erlang consists in not using the concurrence features. This allows to focus on the kernel of Erlang, which consists in basic functional features.

Erlang is **dynamically typed**, that is, there is no need for the programmer to annotate the code with type informations.

The difference with scripting languages, like Bash, Perl, Python or Ruby, is that in these languages, in case of a type error in an operation, there is always a default value which can be implicitly the result.

In Erlang, a type error at run-time ends the execution of the programs, if not handled.

Introduction/Modules

Following a top-down approach, an Erlang program is first structured in **modules**.

At first glance, a module is a file containing Erlang functions.

The name of the module is given in the file by a special directive (i.e., it is not part of the language itself, strictly speaking) and must be the same as the file name on the underlying operating system.

For example, a file `my_prog.erl` must include, as a first statement, the directive

```
-module(my_prog).
```

Note the starting dash and the final period.

Introduction/Shell and virtual machine

Since Erlang modules only contain functions, they serve as a *database of functions* and something else is needed to use them, just like in a relational database one needs a query language.

This role is devoted to the **Erlang shell**, similar to the command line shells on Unix and Windows operating systems, like, respectively, Bash and DOS.

From this shell, the user can open modules and call functions exported by them.

Introduction/Shell and virtual machine (cont)

In order to run the functions, these must be compiled first to **byte-code**, just like Java for instance.

Second, this byte-code is loaded into the **Erlang virtual machine** (called **BEAM**), a task which is hidden from the user and taken care of by the Erlang shell.

Introduction/First module

Consider the file `math2.erl`:

```
-module(math2).
-export([double/1]).

double(X) -> mult(X,2).
mult(X,Y) -> X * Y.
```

This module exports a list of functions consisting of the lone `double` function, which takes only one argument. This function is defined in the body of the module, together with `mult`, which is not visible from the other modules or the Erlang shell.

Note that the order of definition is not meaningful.

Introduction/First compilation

There are two ways of compiling Erlang modules: from the OS shell or from the Erlang shell.

From the OS shell (whose prompt is here represented by the symbol `$`), type

```
$ erl
Erlang (BEAM) emulator version 5.5.2 [source] [async-threads:0] [hipe]

Eshell V5.5.2  (abort with ^G)
1> _
```

The Erlang shell prompt is here "1>", where "1" is the command line number (i.e., the first command is expected).

Introduction/From the Erlang shell

Then, the compilation function `c/1` takes the name of the file to compile, e.g. `math2.erl`, launches the compiler and, if no error is found, the resulting byte-code is output in a file `math2.beam`:

```
1> c(math2).
{ok,math2}
2> _
```

This means that the compilation was successful (`ok`) and the result is the file `math2.beam`. The `/1` in `c/1` is the number of arguments (called **arity**).

Introduction/From the OS shell

The other way is just to call the Erlang compiler from the OS command-line shell:

```
$ erlc math2.erl
$ _
```

Introduction/Running the program

After the module has been compiled to byte-code, we need an Erlang shell to use it.

From the Erlang shell, a function is called by giving the name of the module first, then typing in a colon, the function name and finally its arguments, between parentheses and separated by commas.

For instance

```
1> math2:double(5).
10
2> _
```

If there are no arguments, just type the opening and closing parentheses.

Introduction/A classic

By default, all functions can be recursive. Take a look at the classic factorial mathematical function in Erlang:

```
-module(math1).
-export([fact/1]).

fact(0) -> 1;
fact(N) -> N * fact(N-1).
```

Notice how a function can be defined in a manner similar to the mathematical notation:

$$0! = 1$$
$$n! = n \times (n-1)!$$

Note that the function names in Erlang (e.g. `fact`) must start in lowercase and the variables (e.g. `N`) in uppercase.

Introduction/A classic (cont)

The factorial function is defined by two cases, called **clauses** in Erlang. Clauses of the same function are separated by semicolons and the last one is terminated by a period:

```
...
fact(0) -> ... ; % First clause
fact(N) -> ... . % Second clause
```

The part before the arrow is called the **head** and the part after it until the semicolon or the period is called the **body** of the function:

$$\underbrace{\texttt{fact(N)}}_{\text{head}} \texttt{ -> } \underbrace{\texttt{N * fact(N-1)}}_{\text{body}} \texttt{.}$$

Introduction/Clause selection

The order of the clauses matters.

Given a function call and the clauses of the corresponding function, the Erlang run-time

1. computes the arguments in the call;

2. examines the heads of the clauses, in the order of the module, until the head of a clause **matches** the function call;

3. that matching may bind variables to values in the call (in other words: "the parameters are bound to the arguments") and these variables are replaced by their value in the body of the selected clause;

4. the body is evaluated.

Introduction/Clause selection (cont)

For example, consider the function call `math1:fact(0)`.

We do not need here to evaluate the arguments, since `0` is already a value.

The first clause in module `math1` is

```
fact(0) -> 1
```

Its head is exactly the function call: it is a trivial case of matching. There are no variable in the head, so there is no binding as the result of the matching.

Hence we do not need to replace any variable in the body. Finally, the body, `1`, is the result — there is no need to evaluate it, since it is already a value.

Introduction/Clause selection (cont)

Consider now the function call `math1:fact(1)`.

Matching rule #1: *Two values match if and only if they are equal.*

The head of the first clause is `fact(0)` and `0` does not match `1`, so the first clause is skipped.

The head of the second clause is `fact(N)`.

Matching rule #2: *A variable matches any value.*

Therefore `N` matches `1` and the body of the second clause, `N * fact(N-1)`, is selected. The matching leads to bind `N` to `1`.

Finally, `N` is replaced by `1` in the body, which becomes `1 * fact(1-1)`, and is further evaluated as the result (i.e., the value of the function call).

Introduction/Tracing computations

It is possible to summarise the evaluation of a function call by a series of rewrite steps. For instance

$$\begin{aligned}
\texttt{fact(2)} &\xrightarrow{2} \texttt{2 * fact(2-1)} \\
&\longrightarrow \texttt{2 * fact(1)} \\
&\xrightarrow{2} \texttt{2 * (1 * fact(1-1))} \\
&\longrightarrow \texttt{2 * (1 * fact(0))} \\
&\xrightarrow{1} \texttt{2 * (1 * (1))} \\
&\longrightarrow \texttt{2 * (1)} \\
&\longrightarrow \texttt{2}
\end{aligned}$$

Note: The number on the arrow is the clause number. An arrow with no number is an elementary arithmetic computation.

Introduction/Why order matters

Let us try the following mistake:

```
-module(math1bis).
-export([fact/1]).

fact(N) -> N * fact(N-1);
fact(0) -> 1.
```

Introduction/Why order matters (cont)

Then let us evaluate

$$\begin{aligned}
\texttt{fact(2)} &\xrightarrow{1} \texttt{2 * fact(2-1)} \\
&\longrightarrow \texttt{2 * fact(1)} \\
&\xrightarrow{1} \texttt{2 * (1 * fact(1-1))} \\
&\longrightarrow \texttt{2 * (1 * fact(0))} \\
&\xrightarrow{1} \texttt{2 * (1 * (0 * fact(0-1)))} \\
&\longrightarrow \texttt{2 * (1 * (0 * fact(-1)))} \\
&\xrightarrow{1} \texttt{2 * (1 * (0 * (-1 * fact(-1-1))))} \\
&\longrightarrow \ldots
\end{aligned}$$

This results in an infinite loop because the head of the second clause can never match — and thus terminate the computation.

Introduction/Classic mistakes

Let us try to call an undefined function:

```
1> math:double(5).
** exited: {undef,[{math,double,[5]},
                   {erl_eval,do_apply,5},
                   {shell,exprs,6},
                   {shell,eval_loop,3}]} **

=ERROR REPORT==== 12-Jan-2007::01:46:49 ===
Error in process <0.29.0> with exit value:
{undef,[{math,double,[5]},
{erl_eval,do_apply,5},{shell,exprs,6},{shell,eval_loop,3}]}
```

Similar problem with the call `math2:triple(5)` and `math2:double()` and `math2:mult(5,5)`.

Introduction/Function composition and stop

As in mathematics, function calls can be composed (i.e., nested), as in

```
1> math2:double(math2:double(5)).
20
2> _
```

To stop the Erlang shell, type

```
2> init:stop().
ok
3> $ _
```

Terms/Atoms

Let us define more precisely what kind of data Erlang operates on. These data are called **terms**.

Some terms are only identified by their name, called an **atom**. An atom starts with a lower-case letter which can be followed by a series of characters out of lower-case letters, upper-case letters, digits and the underscore character ('_').

Some atoms can be **quoted** by an opening single-quote and a closing one. In this case, it may contain blank characters. For example

```
anna    apha_beta_proc  x_25    'This is a quoted atom'
x25     call_Java       x_25AB   x_  x____y
```

Function names are atoms.

Terms/Quoted atoms

Quoted atoms look like some strings in some languages, like Bash, but they are very different they cannot be modified. Therefore, they are like constant character strings.

In some programming languages, a character string can be modified in place, but quoted atoms do not allow this.

Terms/Numbers

Numbers. in Erlang include integer numbers and floating-point numbers. The syntax of integer is as expected, for example

```
1    1234    0    -97
```

The lower and larger integers are limited by the actual Prolog system in use.

Floating-point numbers follow the usual syntax too, like

```
100    -7    3.14    -0.06    100.5    1.5e-3
```

Atoms and numbers define the group of **constants**.

Terms/Numbers (cont)

In Erlang, there are no characters. Instead, they can be handled throughout their ASCII code.

There is a special operator `$` which returns the ASCII code for a given character. For example `$A` evaluates to `65`.

It is possible to input integers in a base which is not 10 by using a special notation `#`. For example `16#ffff` represents `65535` (in base 10).

This operator works only if the base ranges from 2 to 16.

Terms/Variables

A **variable** is a name for a term, but, contrary to atoms, variables do not define any object. For example

```
X = 25
```

does not define the constant term `25` but do give it the name `X`. The term `25` is defined just by being written, just like atoms.

One variable denotes a term in a set. For example, in

```
fact(0) -> 1;
fact(N) -> N * fact(N-1).
```

the variable `N` denotes any number which is not the zero integer.

Terms/Variables (cont)

They must start with an upper-case letter and may be followed by any number of letters, digit and underscores, in any order. For example, the following are valid variables:

```
X  Obj_List  Object2  Result  ObjList  X_  Obj__
```

If a variable appears only once in the head of a clause and not in the body, it is an **unknown variable** and can be replaced by an underscore. First, consider

```
-module(bool).
-export([f/2]).
f(true,true)   -> false;
f(true,false)  -> true;
f(false,true)  -> true;
f(false,false) -> false.
```

Terms/Variables (cont)

It is equivalent to

```
-module(bool2).
-export([f/2]).
f(X,X) -> false;
f(X,Y) -> true.
```

and

```
-module(bool1).
-export([f/2]).
f(X,X) -> false;
f(_,_) -> true.
```

Important: When several underscores occur in the same head, each one denotes a different term, in general.

Terms/ Variables/Lexical scoping

Given an occurrence of a variable, the part of the program where this variable is usable, or bound, is called the **scope**.

Erlang uses **lexical scoping**, which means

- the same variable always represents the same term inside a clause;
- the same variable in two different clauses represent different terms, in general.

More about the scope at page 21.

Terms/Tuples

Some terms can be linearly compounded into one term, called a **tuple**.

The number of components of a tuple is called arity. The syntax of tuples is different from mathematics in that it requires curly braces instead of parentheses:

```
{a, 12, 'hello'}
{1, 2, {3, 4}, {a, {b, {c}}}}
{}
```

Note that tuples can

- contain terms of different kinds,
- be embedded,
- be empty.

Terms/Lists

A **list** is a term made of a series of other terms. The terms are separated by commas and enclosed between an opening square bracket and a closing one. For example

```
[1, abc, [12], 'foo bar']
[]
[a, b, c]
```

are lists. Note that terms of different kinds can be mixed. Lists of integers which denote ASCII characters can be represented with a shorthand, e.g. `"abc"` is equivalent to `[$a,$b,$c]`, that is: `[97,98,99]`.

Note that in C++ also, there is no string type.

Terms/Lists (cont)

It is often useful to refer to the first element of a list. That is why Erlang provides a special notation for lists that allows to distinguish the first elements, called the **head**, and to collapse the remaining elements into a list called the **tail**, using the notation `[` *Head* `|` *Tail* `]`.

For example

```
[1, abc, {4,5}, "hello"]
[1, abc, {4,5}, [$h,$e,$l,$l,$o]]
[1, abc, {4,5} | ["hello"]]
[1, abc | [{4,5}, "hello"]]
[1 | [abc, {4,5}, "hello"]]
```

are different ways to denote the same list.

Terms/Lists (cont)

Note that

```
[1, abc, {4,5} | "hello"]
```

is **not** equivalent to the previous lists. It is in fact equivalent to

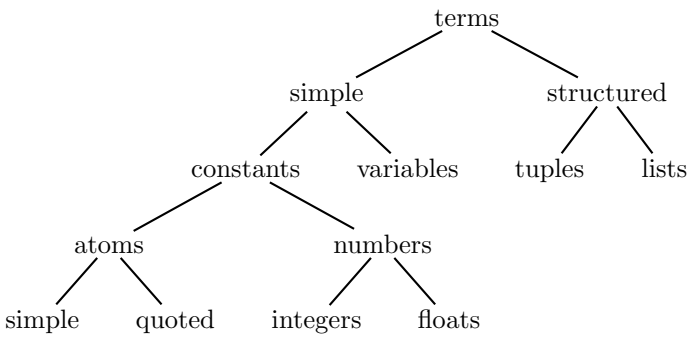
```
[1, abc, {4,5}, $h, $e, $l, $l, $o]
```

Note also that

```
[1, abc | {4,5}, "hello world"]
```

is **invalid** because the tail must be *one* single term.

Terms/Lists (cont)



Ground terms, values and expressions

It is sometimes useful to distinguish between two kinds of terms: **ground terms** and **non-ground terms**.

Ground terms are terms that do not contain any variable.

An **expression** is a syntactic construct involving function calls and terms. For example `5 * fact(5-1)` is an expression but not a term, whereas `X` and `7` are both and expression and a term. The evaluation of an expression does not terminate or lead to a **value** or an error.

A value is thus a ground term that cannot be further computed. For example, `25` is a value but `2 + 5` is not (it is an expression).

Comparing expressions

Erlang offers two different mechanisms to compare expressions: **matching** and numeric comparisons. The former is found in functional and logic programming, but not in mainstream programming languages. The latter is ubiquitous.

Numeric comparisons involve a comparison operator and two operands. For instance, consider the following equality in both Java and Erlang:

```
fact(5) == 119 + 1
```

Note that

1. The two sides are expressions
2. which are separately evaluated,
3. then the values are compared and the result is either true or false.

Java variable definitions

Consider a Java variable definition like

```
int N = 3;
```

This actually means three different things:

1. define the variable `N`
2. whose type is `int`
3. and initial value is `3`.

Initialisation is a separate concept since the above is equivalent to

```
int N;
N = 3;
```

Matching in Erlang

In Erlang, the programmer does not specify the types, as the compiler does not check most of them, so the above is simply written in Erlang

```
N = 3
```

Erlang provides an extension of the Java-like variable definition: *matching*.

1. The left-hand side must be a **term** (not an expression),
2. the right-hand side is an expression
3. which is evaluated before the matching;
4. the matching is itself an expression whose value (in case of success) is the value of the right-hand side (as in C++ "`a = (b = 1);`").

Matching (cont)

What if `N` does not occur for the first time in the matching? Consider

```
-module(match).
-export([f/1]).

f(N) -> N = 3.
```

In this case, `N` in the matching is replaced first by its value, which is the argument of the function `f` when it is called.

So `match:f(3)` succeeds but, for example, `match:f(2)` fails.

Matching (cont)

Therefore, the rule for evaluating the matching

$$t = e$$

is actually better expressed as follows:

1. replace all occurrences of variables in the term t which are already bound by their value: the new term is t';
2. evaluate expression e into v;
3. match v against t';
4. in case of success, the value of the matching is v.

The left-hand side of a matching (here noted t) is called a **pattern**.

Matching (cont)

Assume that `N` appears for the first time in the matching

```
N = 3
```

Then, it is a simple case of matching since

1. the left-hand side, `N`, being a variable, is a term;
2. the right-hand side, `3`, is an expression (actually a value);
3. the evaluation of the right-hand side is itself, `3`, since it is a value already;
4. the matching of numbers is the same as the numeric equality;
5. the value of the matching is `3`.

Importantly, the matching leads to bind variable `N` to `3`, noted $\texttt{N} \mapsto \texttt{3}$.

Matching (cont)

What is the difference, in Erlang, between `N = 3` and `N == 3`?

The first case is a matching and the second is a numeric comparison.

In the latter, `N` must already be defined (bound) so it can be evaluated before the comparison takes place. The result if either the atom `true` or `false`.

In the former, if `N` is already bound, the matching either fails or succeeds and its value is `3`.

If `N` is not already bound, then the matching succeeds with the value `3` and the binding of `N` to `3`, noted $\texttt{N} \mapsto \texttt{3}$.

Matching (cont)

You may wonder now why bother? What is the need of matching when we have equality and other comparisons?

Actually the fist difference is the possible **failure**. Consider

```
-module(match).
-export([f/1]).

f(N) -> N = 3.
```

```
-module(eq).
-export([f/1]).

f(N) -> N == 3.
```

The calls to `match:f` either fail, therefore stop the whole program, or return the integer `3`. The calls to `eq:f` return either `true` or `false`.

Matching (cont)

The main difference between matching and equality appears when matching complex terms, involving lists and tuples. We will explain this from page 22 on.

Remember also that if `N` is not bound, then

```
N = 3
```

acts as a variable definition.

This way, we can think of the matching as a way to combine both definition and comparison.

Matching/Basic examples

Here is another example:

```
N = 3 + 1
```

Here, the right-hand side is not a term, it is evaluated into `4` before the matching takes place.

Another one:

```
N = 2 * f(3)
```

Here, the function `f` must be in the scope, i.e. must be bound at this point.

Scope and multiple clauses

How are variables defined (i.e. bound) by matching used latter? In other words: what is the scope? In C++, the scope is the definition point until the end of the current bloc. For example

```
...
{
  ...
  int n = 3;
  ...  // 'n' is bound here.
}
... // 'n' is unbound here.
```

In Erlang, function bodies can be made of series of expressions separated by commas. For example `f(X) -> X=3, X+1.`

Scope and multiple clauses (cont)

The expressions are evaluated from left to right and, as exemplified in

```
f(X) -> X=3, X+1.
```

the bindings created by matches (a match is a successful matching) are accessible for the following expressions. This is why `X` is bound to the value `3` in the expression `X + 1`.

Note that only the value of the last expression becomes the return value of the function. Thus, for example, the value of `X=3`, which is `3`, is discarded, **but not the binding of `X`!** Remember that evaluating a match yields to

1. a value
2. *and*, if variables are present on the left-hand side, bindings for these variables.

Matching/Constants

Two numbers match if they represent the same mathematical number, e.g.

```
37 = 37
```

Two atoms match if they are made of the same characters, e.g.

```
abc = abc 'hello world' = 'hello world'
```

A variable matches any term that can be evaluated, e.g.

```
X = 37 X = Y
```

Matching/Tuples

A tuple t matches an expression e, that is

$$t = e$$

if and only if

1. the replacement of the variables of t that are bound yields the (term) tuple t';
2. the evaluation of e yields the tuple v;
3. both t' and v have the same arity;
4. the n-th component of t matches the n-th component of v.

In case of success, the resulting bindings of all sub-matches are joined.

Matching/Tuples (cont)

The matching of tuples is very useful, because it allows to destructure the return values of functions in a single expression.

For example

```
go(Job) ->
  {Status,NewJob,Priority} = launch(Job),
  forward(Message),
  schedule(NewJob,Priority).
```

The tuples can be arbitrarily nested:

```
go(Job) ->
  {{status,{Answer,Log}},Version} =
    {run(Job),version(Job)}.
```

Matching/Lists

A list l matches an expression e, that is

$$l = e$$

if and only if

1. the replacement of the variables in l which are bound yields the (term) list l';

2. the evaluation of e yields the list v;
3. both e and v are the empty list `[]` or
4. the head of l' matches the head of v and
5. the tail of l' matches the tail of v.

Matching/Lists/Examples

Consider for instance

```
[Head | Tail] = [1,a,2,b,c]
```

This match yields the value `[1,a,2,b,c]` because of the matches

```
Head = 1
Tail = [a,2,b,c]
```

The bindings are thus `Head` $\mapsto$ `1` and `Tail` $\mapsto$ `[a,2,b,c]`.

Consider also the two matchings

```
{X,Y} = {a,b}, [A,X|B] = [Y,a]
```

which succeed, yielding the bindings `X` $\mapsto$ `a`, `Y` $\mapsto$ `b`, `A` $\mapsto$ `b` and `B` $\mapsto$ `[]`.

Matching/Mixed examples

The matching

```
{C, [Head | Tail]} = {{222, man}, [a,b,c]}
```

succeeds with the bindings {`C` $\mapsto$ {`222, man`}, `Head` $\mapsto$ `a`, `Tail` $\mapsto$ `[b,c]`}.

The matching

```
[{person,Name,Age,_}|T] = [{person,fred,22,male}]
```

succeeds with bindings {`Name` $\mapsto$ `fred`, `Age` $\mapsto$ 22, `T` $\mapsto$ `[]`}. Note the pattern "`_`" which stands for an unknown but unique variable.

Matching/Non-linear patterns

Remember that a matching consists in comparing structurally the value of an expression to a term, called a pattern.

It is possible in Erlang to repeat a variable in a pattern, in order to save another match later. For example

```
{X,X} = {a,f()}
```

is equivalent to

```
{X,Y} = {a,f()}, X = Y
```

The matching

```
{A, foo, A} = {123, foo, abc}
```

fails.

Matching/Calling functions

The Erlang functions are called using pattern matching too. At page 8 we defined the factorial function as

```
-module(math1).
-export([fact/1]).

fact(0) -> 1;
fact(N) -> N * fact(N-1).
```

There are two cases. When evaluating a call, the Erlang virtual machine (called BEAM) needs to determine which case has to be chosen.

This is decided by matching the call against the heads of the cases in turn, here `fact(0)` first then `fact(N)`, considered as patterns — assuming that a function name only matches the same name.

The first match selects the function body to be executed.

Matching/Calling functions (cont)

The procedure can be summarised as follows:

1. evaluate the arguments of the function call;
2. find the cases defining the called function;

3. try to match the head of the first case against the function call;
4. in case of success, the body of the case is evaluated with the bindings of the match;
5. otherwise the next case is tried;
6. if no case matches, an error is issued.

Matching/Calling functions (cont)

Consider a more complex case:

```
X=2, fact(X)
```

1. the argument `X` of `fact(X)` is evaluated with the bindings $\{$`X` $\mapsto$ `2`$\}$, so the call is now `fact(2)`;
2. the matching `fact(0) = fact(2)` fails,
3. so `fact(N) = fact(2)` is tried, and succeeds;
4. the expression `N * fact(N-1)` is evaluated with the bindings $\{$`N` $\mapsto$ `2`$\}$.

Matching/Calling functions (cont)

Consider a function definition relying on the matching of a list:

```
-module(list1).
-export([len/1]).

len([])    -> 0;
len([_|T]) -> 1 + len(T).
```

This function takes a list as argument and return the number of items in it.

There are two cases: one for the empty list and another for the non-empty list.

Function calls

The order in which arguments of function calls are evaluated is undefined (as in C++ but contrary to Java). For instance, in

```
f(g(),h())
```

the order of evaluation of `g()` and `h()` is undefined, i.e.,, the programmer has no guarantee whether `g()` will be evaluated before or after `h()`.

The order of evaluation of elements in a list is also undefined. For instance

```
[g(), h()]
```

Functions/Modules

Modules in Erlang are a database of functions.

The module name must be the basename of the file containing it, and the extension of the file must be `erl`. So, file `foo.erl` contains the Erlang module `foo`, which is specified by the first line

```
-module(foo).
```

The purpose of Erlang modules is to restrict the visibility of the function definitions, since many functions have a merely technical, internal, purpose and are not directly related to the higher-level of module functionalities or services. Exported functions need to be listed, together with their arity. For example

```
-module(foo).
-export([fact/1]).
```

Functions/Qualified calls

Functions can be called in two ways, either by qualifying their name or not. Qualification means writing the name of the module the called function belongs to. For instance

```
math:fact(4)
```

is a qualified call, whereas

```
fact(N-1)
```

is a non-qualified call.

When the caller and the called function are in different modules, the qualified form is mandatory. When calling a function which is in the same module as the caller, the two forms are allowed. There is actually a difference we shall discuss later, but which is, at this point, irrelevant.

Functions/Recursivity

Recursivity consists in a function calling itself, just like the factorial function did at page 8. The function call in the body of the second case is a recursive call.

This technique is the only way in Erlang to implement the equivalent of loops in other imperative programming languages, i.e., in non-functional programming languages.

Indeed, since there are no mutable variables in Erlang, i.e., variables whose value can be changed, a loop would be pointless.

The combination of matching and recursivity is the heart of Erlang programming.