Erlang Introduction 1.

László Lövei

Department of Programing Languages and Compilers
Faculty of Informatics
Eötvös Loránd University

May 21, 2009

Outline

Background

Erlang/OTP has been created by Ericsson *"to provide a better way of programming telephony applications."*

- Highly concurrent: 100,000 simultaneous transactions
- Highly reliable: 99.999% availability
- Soft real-time: react within a certain time
- Distributed over several computers
- Interaction with hardware
- Very large software with complex functionality

Features

- Functional language
  - No destructive assignments
  - Programs consist of function definitions
- Concurrency oriented programming
- Not pure: there are expressions with side effects
- No static type checking
- Features critical for telecom software:
  - Fault tolerance
  - Hot code loading
  - Distributed operation
  - Soft real-time characteristics
  - External interfaces
  - Portability

Function syntax

Simple functions

```
double(Number) -> 2 * Number.
```

- Function identifiers start with a lower case letter

Function syntax

Simple functions

```
double(Number) -> 2 * Number.
```

- Function identifiers start with a lower case letter
- Variable identifiers start with an upper case letter

Function syntax

Simple functions

```
double(Number) -> 2 * Number.
```

- Function identifiers start with a lower case letter
- Variable identifiers start with an upper case letter
- Functions are terminated with a full stop

Function syntax

Simple functions

```
double(Number) -> 2 * Number.
quad(X) -> 2 * double(X).
```

- Function identifiers start with a lower case letter
- Variable identifiers start with an upper case letter
- Functions are terminated with a full stop
- Functions in the same module call each other using their name

Function syntax

Simple functions

```
double(Number) -> 2 * Number.
quad(X) -> 2 * double(X).
hello() -> io:put_chars("Hello!\n").
```

- Function identifiers start with a lower case letter
- Variable identifiers start with an upper case letter
- Functions are terminated with a full stop
- Functions in the same module call each other using their name
- External function calls include a module name qualifier

Pattern matching and guards

Functions may have more clauses, the first clause with a matching pattern and a true guard is executed.

Factorial function

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

- A constant pattern matches only that constant

Pattern matching and guards

Functions may have more clauses, the first clause with a matching pattern and a true guard is executed.

Factorial function

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

- A constant pattern matches only that constant
- A variable pattern binds a value to the variable

Pattern matching and guards

Functions may have more clauses, the first clause with a matching pattern and a true guard is executed.

Factorial function

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

- A constant pattern matches only that constant
- A variable pattern binds a value to the variable
- Other conditions may be specified as guards

Pattern matching and guards

Functions may have more clauses, the first clause with a matching pattern and a true guard is executed.

Factorial function

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

- A constant pattern matches only that constant
- A variable pattern binds a value to the variable
- Other conditions may be specified as guards
- When no clauses are selected, a run time error occurs

Pattern matching and guards

Functions may have more clauses, the first clause with a matching pattern and a true guard is executed.

Factorial function

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

- A constant pattern matches only that constant
- A variable pattern binds a value to the variable
- Other conditions may be specified as guards
- When no clauses are selected, a run time error occurs
- Guards are rather limited to prevent side effects

Other basic concepts

Square root by Newton iteration

```
newton(A) -> newton(A, A).
newton(A, X) -> newton(A, X, (X+A/X)/2).
newton(_, X, Next) when abs(X-Next) < 0.0001 -> Next;
newton(A, _, Next) -> newton(A, Next).
```

- Functions with different arities may have the same name

Other basic concepts

Square root by Newton iteration

```
newton(A) -> newton(A, A).
newton(A, X) -> newton(A, X, (X+A/X)/2).
newton(_, X, Next) when abs(X-Next) < 0.0001 -> Next;
newton(A, _, Next) -> newton(A, Next).
```

- Functions with different arities may have the same name
- Underscore patterns mean “I don’t care about this value”

Other basic concepts

Square root by Newton iteration

```
newton(A) -> newton(A, A).
newton(A, X) -> newton(A, X, (X+A/X)/2).
newton(_A, X, Next) when abs(X-Next) < 0.0001 -> Next;
newton(A, _X, Next) -> newton(A, Next).
```

- Functions with different arities may have the same name
- Underscore patterns mean "I don't care about this value"
- Variables starting with an underscore do not give a warning when unused

Other basic concepts

Square root by Newton iteration

```
newton(A) -> newton(A, A).
newton(A, X) -> newton(A, X, (X+A/X)/2).
newton(_A, X, Next) when abs(X-Next) < 0.0001 -> Next;
newton(A, _X, Next) -> newton(A, Next).
```

- Functions with different arities may have the same name
- Underscore patterns mean "I don't care about this value"
- Variables starting with an underscore do not give a warning when unused
- Unqualified built-in functions are implemented by the emulator

Other basic concepts

Square root by Newton iteration

```
newton(A) when is_float(A);
               is_integer(A) -> newton(A, A).
newton(A, X) -> newton(A, X, (X+A/X)/2).
newton(_A, X, Next) when abs(X-Next) < 0.0001 -> Next;
newton(A, _X, Next) -> newton(A, Next).
```

- Functions with different arities may have the same name
- Underscore patterns mean “I don’t care about this value”
- Variables starting with an underscore do not give a warning when unused
- Unqualified built-in functions are implemented by the emulator
- Some BIFs may be used in guards

Tuples

- Fixed size sequence of arbitrary Erlang data
- Constant time element access by index
- Cannot be modified in any way
- May be empty, upper size is not limited (only by the available memory)
- Syntax: `{El1, El2, ..., ElN}`

Tuple example

There are a number of BIFs for handling tuples:

Complex numbers

```
add(A, B) when is_tuple(A), size(A) == 2,
               is_tuple(B), size(B) == 2 ->
    {element(1, A) + element(1, B),
     element(2, A) + element(2, B)}.

conj(A) when is_tuple(A), size(A) == 2 ->
    setelement(2, A, -element(2, A)).

test() -> add({1, 0}, conj({0, 1})).
```

Tuple example

It is much more common to use pattern matching:

Complex numbers

```
add({ReA, ImA}, {ReB, ImB}) -> {ReA + ReB, ImA + ImB}.
conj({Re, Im}) -> {Re, -Im}.
test() -> add({1, 0}, conj({0, 1})).
```

- A tuple pattern only matches a tuple of the same size
- Elements are also matched recursively

Atoms

- Atoms are character sequences used mainly as labels
- No string operations, only matching
- Function and module names are atoms
- Atoms with funny characters need quotes around them
  - `hello` is the same as `'hello'`
  - `'What\'s this?'` is also an atom

Tagged tuples

Atoms are frequently used to distinguish between different “types”:

File reading

```
read(Name) -> read1(file:read_file(Name)).
read1({ok,     Text})   -> Text;
read1({error, Reason}) -> throw(Reason).
```

- read_file always returns a pair of values

Tagged tuples

Atoms are frequently used to distinguish between different “types”:

File reading

```
read(Name) -> read1(file:read_file(Name)).
read1({ok,     Text})    -> Text;
read1({error, Reason}) -> throw(Reason).
```

- `read_file` always returns a pair of values
- When the first element is `ok`, it means reading has been successful and the contents of the file is returned

Tagged tuples

Atoms are frequently used to distinguish between different “types”:

File reading

```
read(Name) -> read1(file:read_file(Name)).
read1({ok,     Text})   -> Text;
read1({error, Reason}) -> throw(Reason).
```

- `read_file` always returns a pair of values
- When the first element is `ok`, it means reading has been successful and the contents of the file is returned
- `error` means reading has failed

Tagged tuples

Atoms are frequently used to distinguish between different "types":

File reading

```
read(Name) -> read1(file:read_file(Name)).
read1({ok,     Text})   -> Text;
read1({error, Reason}) -> throw(Reason).
```

- `read_file` always returns a pair of values
- When the first element is `ok`, it means reading has been successful and the contents of the file is returned
- `error` means reading has failed
- `throw` inhibits normal function return, and throws an exception that can be caught later

Pattern matching expressions

Branching based on patterns is not restricted to function clauses:

Branching expression

```
case file:read_file(Name) of
   {ok, Text} -> {ok, process(Text)};
   {error, _} -> error
end
```

- The result of the expression is matched on the patterns
- The first clause with a matching pattern is executed
- Guards can be used as well

Pattern matching expressions

Pattern matching may be used without branching as well:

Simple pattern match

```
process_file(Name) ->
   {ok, Text} = file:read_file(Name),
   process(Text).
```

- Works as an assertion
- Generates a run time error when the pattern does not match
- Alternative (error prone) style:

  `Text = element(2, file:read_file(Name))`

Lists

- Traditional functional lists built using `[Head|Tail]` and `[]`
- A single list cell cannot be modified, but building a new list by prepending an element is very efficient
- Better suited to storing variable length data than tuples in spite of linear time element access
- Syntactic sugar:
  - `[El1, ..., ElN]` means `[El1, [..., [ElN|[]]]]`
  - `[El1, El2 | Tail]` can also be used
- BIFs: `length`, `hd`, `tl`

Sum of numbers

Selector style

```
sum(L) when L == [] -> 0;
sum(L)              -> hd(L) + tl(L).
```

Pattern matching style

```
sum([])      -> 0;
sum([Hd|Tl]) -> Hd + sum(Tl).
```

- The latter is preferred

Special types

Fun
: Unnamed function (lambda expression)

Binary
: A sequence of uninterpreted bytes
  - Special syntax for pattern matching
  - Sometimes used to store strings

Pid
: Identifier for Erlang processes

Port
: Identifier for an external connection (e.g. hardware driver)

Ref
: An opaque identifier uniquely generated by `make_ref`

Strings

- The canonical representation is a list of integers (character codes)
- Syntactic sugar:
  - `"ABC"` means `[65,66,67]`
  - `[$A, $B, $C]` means the same
- Extensive library support (modules `lists` and `strings`)
- Deep strings: `["A", ["BC", ["D"], "E"]]`
  - Efficient concatenation
  - Library support: the `io` module prints it as `ABCDE`
  - `lists:flatten` converts it to flat string
- String representation of Erlang data: `io_lib:format`
  - `io_lib:format("~p", [AnyData])`
  - `io_lib:format("~b, ~f, ~c", [Int, Float, Char])`
  - Direct printing: `io:format("~s~n", [TextOrAtom])`

Booleans

- Conventional representation: atoms `true` and `false`
- Comparison operators (`==`, `/=`, `=:=`, `=/=`, `<`, `>`, `=<`, `>=`) return these atoms
- Boolean operators expect and return these atoms (`and`, `or`)
- Library functions use these atoms (e.g. `lists:any`, `lists:all`, `lists:filter`)
- Shortcut boolean operators: `andalso`, `orelse`
  - The second argument may be anything, it is simply returned

Module syntax

complex.erl

```
-module(complex).
-export([add/2, conj/1, test/0]).
add({ReA, ImA}, {ReB, ImB}) -> {ReA + ReB, ImA + ImB}.
conj({Re, Im}) -> {Re, -Im}.
test() -> add({1, 0}, conj({0, 1})).
```

- Module name must match the file name

Module syntax

complex.erl

```
-module(complex).
-export([add/2, conj/1, test/0]).
add({ReA, ImA}, {ReB, ImB}) -> {ReA + ReB, ImA + ImB}.
conj({Re, Im}) -> {Re, -Im}.
test() -> add({1, 0}, conj({0, 1})).
```

- Module name must match the file name
- Only the exported functions may be called externally

Module syntax

complex.erl

```
-module(complex).
-export([add/2, conj/1, test/0]).
add({ReA, ImA}, {ReB, ImB}) -> {ReA + ReB, ImA + ImB}.
conj({Re, Im}) -> {Re, -Im}.
test() -> add({1, 0}, conj({0, 1})).
```

- Module name must match the file name
- Only the exported functions may be called externally
- Every attribute and function is terminated by a full stop

The Erlang shell

- Started by `erl` (Unix) or `werl` (Windows)
- Evaluates expressions interactively
- Functions and modules cannot be defined on the fly
- Compilation and module loading easily accessible
- Many tools can be started from the shell: graphical debugger, process monitor, profiler, error analyser, documentation generator, etc.

Example session

```
$ erl
Erlang (BEAM) emulator version 5.6.3 [source] [hipe] ...

Eshell V5.6.3  (abort with ^G)
1> c(complex).
ok,complex
2> complex:test().
{1,-1}
3> halt().
```

1. Compile and load `complex.erl`

Example session

```
$ erl
Erlang (BEAM) emulator version 5.6.3 [source] [hipe] ...

Eshell V5.6.3  (abort with ^G)
1> c(complex).
ok,complex
2> complex:test().
{1,-1}
3> halt().
```

1. Compile and load `complex.erl`
2. Call a function, the return value is displayed

Example session

```
$ erl
Erlang (BEAM) emulator version 5.6.3 [source] [hipe] ...

Eshell V5.6.3  (abort with ^G)
1> c(complex).
ok,complex
2> complex:test().
{1,-1}
3> halt().
```

1. Compile and load `complex.erl`
2. Call a function, the return value is displayed
3. Stop the emulator

Useful shell commands

- `help().` gives help
- `cd(Path).` changes the working directory
- `pwd().` prints the working directory
- `ls().` lists the files in the working directory
- `v(N).` returns the result of the n^{th} expression
- `f(V).` clears the binding of shell variable `V`
- `f().` clears the binding of every shell variable

Exercises

Write an Erlang function that...

1. calculates the n^{th} Fibonacci number (try large numbers!)
2. returns the maximal element from a list of integers
3. counts the words in a string
4. calculates every Pythagorean triple below a given limit
5. converts the upper case letters to lower case in a string
6. calculates the first n rows of Pascal's triangle