Erlang – functional kernel

Paulo Sérgio Almeida

Grupo de Sistemas Distribuídos
Departamento de Informática
Universidade do Minho

Functional kernel

- data types;
- variables;
- pattern matching;
- functions;
- guards;
- sequences;
- conditionals;
- tail calls and state machines;
- modules;
- higher-order functions;
- comprehensions;
- binaries and bitstreams;

Simple data types

- Numbers:
  - integers (arbitrary size) or floating point;

```
123, 3.1415, -1.2e3
2#11101001, 16#10FE       % base 2 and 16; bases between 2 and 16
$A, $\n                       % ascii value of A and \n
```

- Atoms:
  - named constants;
  - start by lower case; single quotes to use special chars;

```
true, false         % booleans
red, green, blue, request, reply, 'hello world\n'
```

- Pids:
  - store process identifiers;
  - can be used as message destination
- References:
  - store references that are globally unique in the system;

Composite data types

- Tuples: fixed number of items;

```
{add, 123}
{}
```

- Lists: variable number of items; polymorphic;

```
[]
[1, 2, 3]
[1, 1.1, abc]
[[1,2], abc, [1, a]]
[1 | [2, 3]]    % = [1,2,3]
"ABCD"          % = [65,66,67,68]; there are no strings in Erlang
```

- Maps: key to value mappings

```
#{}
#{name => "Alice", age => 25, friends => ["Bob", "Carol"]}
#{[2,4] => {circle, 3}, [9, 7] => {rectangle, 3, 5}}
```

Variables

- Start by upper case;

```
X = 2
Y = {3, abc}
Z = {color, [{red, 0.45}, {blue, 0.23}, {blue, 0.8}]}
```

- They are not variables, but bindings of values to names;
- Single assignment: after bound, they cannot be updated;

Pattern matching

- Used to bind a value to a pattern;
  - in assignment to variables;
  - binding arguments to parameters of a function;
  - in the “case” expression;
  - delivering messages in a receive;
- Examples:

```
A = 12
{X, Y} = {2, 3}
{C, D} = {[1,2,3], {hello, world}}      % C=[1,2,3]  D={hello, world}
[H|T] = [1,2,3]                         % H=1  T=[2,3]
{ok, Res} = {ok, 23}                    % Res=23
{_, Res} = {ok, 23}                     % Res=23  _ matches and ignores
[A,A,C] = [1,1,3]                       % A=1  C=3
[A,A,C] = [1,2,3]                       % Fails
{ok, Res} = {red, 23}                   % Fails
[A,B,C,D] = [1,2,3]                     % Fails
Color = #{ r => 0.2, g => 0.5, b => 0.7}
#{r := Red, b := Blue} = Color          % Red = 0.2, Blue = 0.7
```

Functions

- Can be defined by clauses, with pattern matching and recursion:

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

- Clauses are separated by ';'; definition ends with '.';
- The first clause that matches is used;
- In the example above order matters; in the next one it doesn't:

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

Guards

- Can be used in functions, conditionals, and receive;

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

- A guard sequence is a sequence of guards separated by ';'; the guard sequence succeeds if one of the guards do so;
- A guard is a sequence of tests separatated by commas; all tests must be true for the guard to succeed;
- Tests can contain constants, arithmetic expressions, comparisons and some pre-defined tests such as:

```
is_atom/1, is_binary/1, is_constant/1, is_float/1, is_function/1,
is_function/2, is_integer/1, is_list/1, is_number/1, is_pid/1,
is_port/1, is_reference/1, is_tuple/1, is_record/2, is_record/3
```

and some pre-defined functions as:

```
abs(Number), element(N, Tuple), float(Term), hd(List),
length(List), node(), node(Pid|Ref|Port), round(Number),
self(), size(Tuple|Binary), tl(List), trunc(Number)
```

Sequences

- The body of a clause is a sequence of expressions;
- Expressions are separated by commas;
- Expressions are evaluated sequentially;
  - relevant if they have side-effects such as sending messages;

```
factorial(N) when N > 0 ->
  N1 = N - 1,
  F1 = factorial(N1),
  N * F1;
factorial(0) ->
  1.
```

Conditional expressions – if

```
if
  GuardSeq1 -> Seq1;
  GuardSeq2 -> Seq2;
  ...
end
```

- If no guard succeeds the process crashes;
- There is no 'else'; 'true' can be used in the last guard:

```
if
  ...
  true -> ...
end
```

- Example:

```
factorial(N) ->
  if
    N == 0 -> 1;
    N >  0 -> N * factorial(N-1)
  end.
```

This factorial is not defined for negative numbers.

Conditional expressions – case

```
case Expr of
  Pattern1 [when GuardSeq1] -> Seq1;
  ...
  PatternN [when GuardSeqN] -> SeqN
end
```

- Example:

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

Tail recursion

- In Erlang, loops are obtained using recursion;
- Tail calls, in which the recursive invocation is at the end, are specially important;
- Avoids stack growth, running in constant memory;
- Important for processes that keep running indefinitely;
- Example; compare:

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

with (using name overloading in len for 1 and 2 parameters):

```
len(L) ->
  len(L,0).

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

State machines with tail calls

- In general, several functions may invoke each other recursively;
- If they use tail calls they execute in constant memory;
- Tail calls allow easily implementing a state machine, with each function representing a state or family of states;

```
acquired(X) ->
  ...
  released(Y).

released(X) ->
  ...
  acquiring(Y).

acquiring(X) ->
  receive
    ...
      acquiring(Y);
    ...
      acquired(Z)
  end.
```

Modules

- A program is composed of modules, that define functions;
- Names of modules and functions are atoms;
- Functions used outside a module must be exported;
- Example:
  - module demo, exports factorial function with 1 parameter;
  - at beginning of demo.erl file:

```
-module(demo).
-export([factorial/1]).
```

  - using factorial in other module:

```
demo:factorial(3)
```

  - or using import (although import is not commonly used)

```
-import(demo, [factorial/1]).

factorial(3)
```

Higher-order functions

- Anonymous functions can be created with fun;
- Example: adder returns function that adds N to argument:

```
adder(N) ->
  fun (X) -> X + N end.
```

- The closure references N, which is bound when invoking adder;
- The function returned may be invoked using normal syntax;

```
F = adder(2),
F(3).           % = 3+2
```

- A global (named) function may be passed to a context that requires a closure with:

```
fun name_func/arity
fun module:name_func/arity
```

- Example:

```
lists:filter(fun even/1, lists:seq(1,10))
```

- Module `lists` has traditional functions like: `map`, `foldl`, `zip`, `filter`, ...

```
L = lists:seq(1, 100),
IsEven = fun(X) ->
           if X rem 2 == 0 -> true;
             true -> false
           end
         end,
EvenNumbers = lists:filter(IsEven, L),
lists:map(fun(X) -> 3 * X end, EvenNumbers).
```

- The above can be obtained using a list comprehension:

```
[ 3 * X || X <- lists:seq(1, 100), X rem 2 == 0]
```

Binaries

- Untyped block of memory (sequence of bytes);
- Used to process files, streams;
- Built using bit-syntax;
- Traversed using pattern-matching;
- Example: function that receives and parses an IPv4 datagram using pattern-matching:

```
parse_IP_packet(Dgram) ->
  DgramSize = size(Dgram),
  case Dgram of
    <<?IP_VERSION:4, HLen:4, SrvcType:8, TotLen:16,
      ID:16, Flgs:3, FragOff:13,
      TTL:8, Proto:8, HdrChkSum:16,
      SrcIP:32, DestIP:32, RestDgram/bytes>>
    when HLen >= 5, 4*HLen =< DgramSize ->
      OptsLen = 4*(HLen - ?IP_MIN_HDR_LEN),
      <<Opts:OptsLen/bytes,Data/bytes>> = RestDgram,
      ...
    end.
```

Bitstreams

- Bitstream: sequence of bits
- Generalization of binaries: no need to be a multiple of 8 bits;
- Efficient processing of streams that are not byte oriented;
- UU-encoding in one line, with bitstream comprehension:

```
uuencode(BitStr) -> << <<(X+32):8>> || <<X:6>> <= BitStr >>.

uudecode(Text) -> << <<(X-32):6>> || <<X:8>> <= Text >>.
```

- Comprehensions can iterate over or create both lists or bitstreams, with the 4 combinations possible.