Control Systems Optimization

Igor Wojnicki

AGH – Univeristy of Science and Technology

2010

Outline

Credits

- http://www.erlang.org
- http://learnyousomeerlang.com/
- http://erlang.org/course/course.html

Features

- Erlang/OTP (Open Telecom Platform).
- Soft real-time.
- References
  - Ericsson telephone switching systems.
  - Facebook chat.
  - CouchDB.
  - Mobilearts GSM/UMTS services.
  - T-Mobile

Erlang Suitability

- Telecommunication systems, e.g. controlling a switch or converting protocols.
- Servers for Internet applications, e.g. a mail transfer agent, an IMAP-4 server, an HTTP server or a WAP Stack.
- Telecommunication applications, e.g. handling mobility in a mobile network or providing unified messaging.
- Database applications which require soft realtime behaviour.

Architecture

- Source code → erl
- Compiler → beam
- Virtual Machine
- Shell

Main Characteristics

- Everything is a function – returns a value
- Multiple process.
- IPC: messages – no shared resources, no deadlock possible.

Starting the System

```
$ erl
1> c(demo).
{ok,demo}
2> demo:double(25).
50
3> demo:times(4,3).
** exception error: undefined function demo:times[4,3]
4> 10 + 25.
35
5>
```


- `c(File)` compiles the file `File.erl`.
- `1>` , `2>` ... are the shell prompts.
- The shell sits in a read-eval-print loop.

Shell Commands

`h()` - history . Print the last 20 commands.
`b()` - bindings. See all variable bindings.
`f()` - forget. Forget all variable bindings.
`f(Var)` - forget. Forget the binding of variable X. This can ONLY be used as a command to the shell - NOT in the body of a function!
`e(n)` - evaluate. Evaluate the n:th command in history.
`e(-1)` - Evaluate the previous command.
`help()` - Available commands.

- Edit the command line as in Emacs
- See the User Guide for more details and examples of use of the shell.

Shell (cont)

- ^C BREAK: (a)bort (c)ontinue (p)roc info (i)nfo (l)oaded (v)ersion (k)ill (D)b-tables (d)istribution
- ^G User switch command

```
--> h
 c [nn]                - connect to job
 i [nn]                - interrupt job
 k [nn]                - kill job
 j                     - list all jobs
 s [shell]             - start local shell
 r [node [shell]]      - start remote shell
 q          - quit erlang
 ? | h                 - this message
```

Numbers

- Integers

```
10
-234
16#AB10F
2#110111010
$A
```

- Floats

```
17.368
-56.654
12.34E-10.
```

- B#Val is used to store numbers in base B
- $Char is used for ascii values (example $A instead of 65).

Atoms

```
abcef
start_with_a_lower_case_letter
'Blanks can be quoted'
'Anything inside quotes \n\012'
```

- Indefinite length atoms are allowed.
- Any character code is allowed within an atom.

Tuples

```
{123, bcd}
{123, def, abc}
{person, 'Joe', 'Armstrong'}
{abc, {def, 123}, jkl}
{}
```

- Used to store a fixed number of items.
- Tuples of any size are allowed.

Lists

```
[123, xyz]
[123, def, abc]
[{person, 'Joe', 'Armstrong'},
     {person, 'Robert', 'Virding'},
     {person, 'Mike', 'Williams'}
]
abcdefghi becomes [97,98,99,100,101,102,103,104,105]
"" becomes []
```


- Used to store a variable number of items.
- Lists are dynamically sized.
- "..." is short for the list of integers representing the ascii character codes of the enclosed within the quotes.

Variables

```
Abc
A_long_variable_name
AnObjectOrientatedVariableName
```

- Start with an Upper Case Letter.
- No “funny characters”.
- Variables are used to store values of data structures.
- Variables can only be bound once! The value of a variable can never be changed once it has been set (bound).

Complex Data Structures

```
[{{person,'Joe', 'Armstrong'},
        {telephoneNumber, [3,5,9,7]},
        {shoeSize, 42},
        {pets, [{cat, tubby},{cat, tiger}]},
        {children,[{thomas, 5},{claire,1}]}},
 {{person,'Mike','Williams'},
        {shoeSize,41},
        {likes,[boats, beer]},
        ...
```


- Arbitrary complex structures can be created.
- Data structures are created by writing them down (no explicit memory allocation or deallocation is needed etc.).
- Data structures may contain bound variables.

Pattern Matching

- Note the use of "_", the anonymous (don't care) variable.

```
A = 10
```

Succeeds – binds A to 10

```
{B, C, D} = {10, foo, bar}
```

Succeeds – binds B to 10, C to foo and D to bar

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

Succeeds – binds A to abc, B to foo

```
{A, A, B} = {abc, def, 123}
```

Fails

```
[A,B,C] = [1,2,3]
```

Succeeds – binds A to 1, B to 2, C to 3

```
[A,B,C,D] = [1,2,3]
```

Fails

Pattern Matching (Cont)

`[A,B|C] = [1,2,3,4,5,6,7]` Succeeds – binds A = 1, B = 2, C = [3,4,5,6,7]

`[H|T] = [1,2,3,4]` Succeeds – binds H = 1, T = [2,3,4]

`[H|T] = [abc]` Succeeds – binds H = abc, T = []

`[H|T] = []` Fails

`{A,_, [B|_],{B}} = {abc,23,[22,x],{22}}` Succeeds – binds A = abc, B = 22

Function Calls

```
module:func(Arg1, Arg2, ... Argn)

func(Arg1, Arg2, .. Argn)
```

- Arg1 .. Argn are any Erlang data structures.
- The function and module names (func and module in the above) must be atoms.
- A function can have zero arguments. (e.g. date() – returns the current date).
- Functions are defined within Modules.
- Functions must be exported before they can be called from outside the module where they are defined.

Module System

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

double(X) ->
        times(X, 2).

times(X, N) ->
        X * N.
```


- `double` can be called from outside the module, `times` is local to the module.
- `double/1` means the function double with one argument (Note that `double/1` and `double/2` are two different functions).

Built In Functions (BIFs)

```
date()
time()
length([1,2,3,4,5])
size({a,b,c})
atom_to_list(an_atom)
list_to_tuple([1,2,3,4])
integer_to_list(2234)
tuple_to_list({})
```


- Are in the module erlang.
- Do what you cannot do (or is difficult to do) in Erlang.
- Modify the behaviour of the system.
- Described in the BIFs manual.

Function Syntax

Is defined as a collection of clauses.

```
func(Pattern1, Pattern2, ...) ->
        ... ;
func(Pattern1, Pattern2, ...) ->
        ... ;
        ...
func(Pattern1, Pattern2, ...) ->
        ... .
```

Evaluation Rules

- When a match is found all variables occurring in the head become bound.
- Clauses are scanned sequentially until a match is found.
- Variables are local to each clause, and are allocated and deallocated automatically.
- The body is evaluated sequentially.

Functions – Examples I

```
-module(mathStuff).
-export([factorial/1, area/1]).

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

Functions – Examples II

```
area({square, Side}) ->
        Side * Side;
area({circle, Radius}) ->
        % almost :-)
        3 * Radius * Radius;
area({triangle, A, B, C}) ->
        S = (A + B + C)/2,
        math:sqrt(S*(S-A)*(S-B)*(S-C));
area(Other) ->
        {invalid_object, Other}.
```

Evaluation Example

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

> factorial(3)
        matches N = 3 in clause 2
        == 3 * factorial(3 - 1)
        == 3 * factorial(2)
        matches N =2 in clause 2                    == 3
        == 3 * 2 * factorial(1)
        matches N = 1 in clause 2
        == 3 * 2 * 1 * factorial(1 - 1)
        == 3 * 2 * 1  * factorial(0)
        == 3 * 2 * 1  * 1 (clause 1)
        == 6
```

Variables are local to each clause; allocated/deallocated automatically.

Guarded Function Clauses

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

- The reserved word when introduces a guard.
- Fully guarded clauses can be re-ordered.

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

- This is NOT the same as:

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

- (incorrect!!)

Examples of Guards

is_number(X)	X is a number
is_integer(X)	X is an integer
is_float(X)	X is a float
is_atom(X)	X is an atom
is_tuple(X)	X is a tuple
is_list(X)	X is a list
length(X) == 3	X is a list of length 3
size(X) == 2	X is a tuple of size 2.
X > Y + Z	X is > Y + Z
X == Y	X is equal to Y
X =:= Y	X is exactly equal to Y
(i.e. 1 == 1.0 succeeds but 1 =:= 1.0 fails)	

- All variables in a guard must be bound.

Traversing Lists

```
average(X) -> sum(X) / len(X).

sum([H|T]) -> H + sum(T);
sum([]) -> 0.

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


- Note the pattern of recursion is the same in both cases. This pattern is very common.

Traversing Lists (cont)

Two other common patterns:

```
double([H|T]) -> [2*H|double(T)];
double([]) -> [].

member(H, [H|_]) -> true;
member(H, [_|T]) -> member(H, T);
member(_, []) -> false.
```

Lists and Accumulators

```
average(X) -> average(X, 0, 0).

average([H|T], Length, Sum) ->
        average(T, Length + 1, Sum + H);
average([], Length, Sum) ->
        Sum / Length.
```


- Only traverses the list ONCE
- Executes in constant space (tail recursive)
- The variables `Length` and `Sum` play the role of accumulators
- N.B. `average([])` is not defined - (you cannot have the average of zero elements) – evaluating average([]) would cause a run-time error.

Special Functions

```
apply(Mod, Func, Args)
```

- Apply the function Func in the module Mod to the arguments in the list Args.
- Mod and Func must be atoms (or expressions which evaluate to atoms).

```
1> apply( lists1,min_max,[[4,1,7,3,9,10]]).
{1, 10}
```

- Any Erlang expression can be used in the arguments to apply.

Special Forms

```
case lists:member(a, X) of
        true ->
                ... ;
        false ->
                ...
end,
...

if
        integer(X) -> ... ;
        tuple(X) -> ...
end,
...
```


- Not really needed - but useful.

Stopping VM

```
init:stop().
```

Running From a Command Line

```
erl -noshell -s hello hello_world -s init stop
```