Control Systems Optimization

Igor Wojnicki

AGH – Univeristy of Science and Technology

2010

Outline

1. Erlang: Binaries, Communication with OS

Bit Syntax

- Another data type: **binary** – a set of **bytes**. «"cat"» «99,97,116»
- Functions to manipulate Bits:
  - list_to_binary/1
  - split_binary/2 - splits at a given offset 1>
    split_binary(«1,2,3,4,5,6,7,8,9,10», 3).
    {«1,2,3»,«4,5,6,7,8,9,10»}
  - size/1 – size in bytes

Sub-byte Computations: Packing Data

- Heavy use of pattern matching makes it simple.

```
1> Red = 2.
2
2> Green = 61.
61
3> Blue = 20.
20
4> Mem = <<Red:5, Green:6, Blue:5>>.
<<23,180>>
```

Sub-byte Computations: Unpacking Data

```
5> <<R1:5, G1:6, B1:5>> = Mem.
<<23,180>>
6> R1.
2
7> G1.
61
8> B1.
20
```

Bins in General

```
<<>>
<<E1, E2, ..., En>>

Ei = Value |
Value:Size |
Value/TypeSpecifierList |
Value:Size/TypeSpecifierList
```

- Total number of buts dvisible by 8 !!!
- `Value` – abound variable.
- `Size` – expression evaluating to integer.
- `TypeSpecifierList` : `End-Sign-Type-Unit`
  - `End=big|little|native`
  - `Sign=signed|unsigned`
  - `Type = integer|float|binary` 8|64|any number of bits respectively
  - `Unit = 1 | 2 | ... 255`

Bins-of-Unknown-Size Processing.

```
28> B=list_to_binary([254,1,2,3,4,5,6,7,8,9,10]).
<<254,1,2,3,4,5,6,7,8,9,10>>
29> <<254,1,2,Rest/binary>>=B.
<<254,1,2,3,4,5,6,7,8,9,10>>
30> Rest.
<<3,4,5,6,7,8,9,10>>
31> <<_:7,X:1,Other/binary>>=B.
<<254,1,2,3,4,5,6,7,8,9,10>>
32> X.
0
33> Other.
<<1,2,3,4,5,6,7,8,9,10>>
```

Bins and Files I

- Files can be read as Bins.

```
r2(Filename,String) when is_binary(String) ->
    case file:open(Filename,[read,binary,raw,read_ahead]) of
        {ok,FH} ->
            Val=read_buffer2(FH,String,size(String),
                             32*1024,0,[]), % 32k buffer
            file:close(FH),
            Val;
        Err -> {Filename,Err}
    end.
```

Bins and Files II

```
read_buffer2(FH,String,Len,Buffer_size,Pos,Found) ->
     case file:pread(FH,Pos,Buffer_size) of
         {ok,Buffer} ->
             New_buffer_size=byte_size(Buffer),
             Found_new=process_buffer2(Buffer,New_buffer_size,
                                       String,Len,Pos,Found),
             read_buffer2(FH,String,Len,Buffer_size,
                          Pos+Buffer_size,Found_new);
         Err -> {Pos,[Err|Found]}
     end.
```

Bins and Files III

```
process_buffer2(Buffer, Buffer_size, String, Len, File_pos,
                Found) when Buffer_size >= Len ->
    case re:run(Buffer,String,[global]) of
        {match,List} ->
            Map=lists:map(fun([{Idx,_}]) -> File_pos+Idx end,
                          List),
            [Map|Found];
        nomatch -> Found
    end.
```

Bins Comprehension

- Similar to the List Comprehensions.

```
1> [ X || <<X>> <= <<1,2,3,4,5>>, X rem 2 == 0].
[2,4]
2> Pixels = <<213,45,132,64,76,32,76,0,0,234,32,15>>.
<<213,45,132,64,76,32,76,0,0,234,32,15>>
```

- Binaries and Lists

```
3> RGB = [ {R,G,B} || <<R:8,G:8,B:8>> <= Pixels ].
[{213,45,132},{64,76,32},{76,0,0},{234,32,15}]
```

- Lists and Binaries

```
4> << <<R:8, G:8, B:8>> ||  {R,G,B} <- RGB >>.
<<213,45,132,64,76,32,76,0,0,234,32,15>>
```

Defining Own Control Abstractions

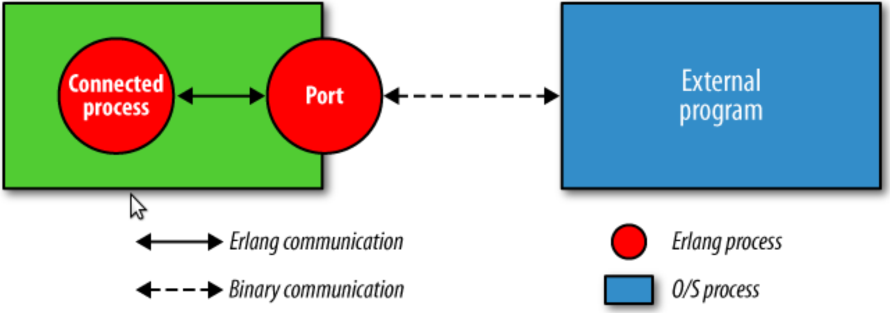
```
for(Max, Max, F) -> [F(Max)];
for(I, Max, F) -> [F(I)|for(I+1, Max, F)].
```

- Simple examples

```
1> lib_misc:for(1,10,fun(I) -> I end).
[1,2,3,4,5,6,7,8,9,10]
2> lib_misc:for(1,10,fun(I) -> I*I end).
[1,4,9,16,25,36,49,64,81,100]
```

Ports

- Communication with OS (other languages)



Ports: a Common Sequence of Operations

```
Port = open_port({spawn, Cmd}, ...),
...
port_command(Port, Payload),
...
receive
{Port, {data, Data}} ->
```

Ports

- Opening a port:
  `Port = open_port(PortName, PortSettings)`
- Communication via messages: Send Data (an IO list) to the port.
  `Port ! {PidC, {command, Data}}`
  Change the PID of the connected process from PidC to Pid1.
  `Port ! {PidC, {connect, Pid1}}`
  Close the port.
  `Port ! {PidC, close}`
- or a function call:
  `port_command(Port, Data)`

Ports, Example, C

```
#include <stdio.h>
#include <stdlib.h>

int main(void){
  char buff[256];
  for (;;) {
    scanf("%255s",buff);
    if (strcmp(buff,"quit")==0) break;
    printf("This is prg.c talking, received:%s",buff);
    fflush(0);
  };
}
```

Ports, Example, Erlang

```
-module(prg).
-export([start_prg/0,prg/2]).

start_prg() ->
    open_port({spawn,"./prg"},[binary, exit_status]).

prg(Port,Msg) ->
    port_command(Port,Msg),
    receive
        {Port, {data, Payload}} ->
            Payload;
        Error -> Error
    after
        1000 -> timeout
    end.
```

Ports, Packet Based Communication, C I

```
#include <stdio.h>
#include <stdlib.h>

int main(void){
  char buff[256];
  int length;
  int i;
  int c;

  for (;;) {
    length=getchar();
    /*    fprintf(stderr,"length: %d\n",length);*/
    if (length==EOF) exit(1);
    for (i=0; i<length; i++){
```

Ports, Packet Based Communication, C II

```
      c=getchar();
      /*      fprintf(stderr,"read %d: %c\n",i,c);*/
      if (c==EOF) exit(1);
      buff[i]=c;
    }
    buff[length]=0;
    if (strcmp(buff,"quit")==0) break;
    printf("%c%s",length,buff);
    fflush(0);
    /*    fprintf(stderr,"there"); */
  };
  return 0;
}
```

Ports, Packet Based Communication, Erlang

```
-module(prg1).
-export([start_prg/0,prg/2]).

start_prg() ->
    open_port({spawn,"./prg1"},[binary, exit_status,
                                {packet,1}]).


prg(Port,Msg) ->
    port_command(Port,Msg),
    receive
        {Port, {data, Payload}} ->
            Payload;
        {Port, {exit_status, S}} -> {exit,S}
    after
        1000 -> timeout
    end.
```

Ports: a Separate Process to Handle a Port I

```
-module(prg2).
-export([start/0,stop/0,call_port/1]).

start() ->
    spawn(fun() ->
                  register(prg2, self()),
                  process_flag(trap_exit, true),
                  Port = open_port({spawn, "./prg1" },
                         [binary, exit_status, {packet,1}]),
                  loop(Port)
          end).

stop() ->
    prg2 ! stop.
```

Ports: a Separate Process to Handle a Port II

```
call_port(Msg) ->
    prg2 ! {call, self(), Msg},
    receive
        {prg2, Result} ->
            Result
    end.

loop(Port) ->
    receive
        {call, Caller, Msg} ->
            Port ! {self(), {command, Msg}},
            receive
                {Port, {data, Data}} ->
                    Caller ! {prg2, Data};
```

Ports: a Separate Process to Handle a Port III

```
      {Port, {exit_status, S}} ->
        Caller ! {prg2, {exit,S}}
    end,
    loop(Port);
  stop ->
    Port ! {self(), close},
    receive
      {Port, closed} ->
        exit(normal)
    end;
  {'EXIT', Port, Reason} ->
    exit({port_terminated,Reason})
end.
```