Sequential Erlang

Lectures F4 + F5 (Chapters 7..10)

Joe Armstrong

Ericsson AB

F4 - Sequential Erlang 2(a)

- Chapters 7,8
- Binaries - storing blocks.
- Bit Syntax - manipulation bits.
- Bit Syntax examples.
- Guards - extending pattern matching.
- Rest of Sequential Erlang - lots of smll things.

Why Binaries and the Bit Syntax?

- Representing large blogs of data.

- Parsing Protocol data

```
|----------+------------+------+---------+-----|
| DataRate | FlowNumber | Port | SubRack | ... |
|----------+------------+------+---------+-----|
 3 bits      5 bits     2 bits     7 bits
```

- In Erlang this is easy:

```
<<DataRate:3, FlowNumber:5,Port:2,...>>
```

Why the bit syntax?

- Designed for packing and unpacking bit aligned data.
- Very efficient.
- Packing and unpacking bit aligned data with bsr,bsl,xor,band is tedious and extremely error prone.
- No other language has this.
- Origonally designed for implementing S7 signalling (and related protocols).

Binaries

- Used to store large data “blobs”.
- 1 byte / per byte + a small overhead – lists are 8 bytes/byte.
- Not copied in inter-process message passing *on the same machine*.
- “fancy” GC.

Binaries - syntax

```
1> X = <<5,10,22>>.
<<5,10,22>>
2> B = <<45,X/binary>>.
<<45,5,10,22>>
```

X= 00000101, 00001010, 00010110

Endianness of integers

- big endian - most significant byte first.
- little endian - most signicant byte last.
- network byte order – same as big.
- You should always encode integers as Big Endian integers in network protocols.

Packing/Unpacking Binaries

```
1> <<2#0000000100000001:16,2>>.
<<1,1,2>>
2> <<1234:32/big,2>>.
<<0,0,4,210,2>>
3> <<1234:32/little,2>>.
<<210,4,0,0,2>>
4> <<1234:32/native,2>>.
<<210,4,0,0,2>>
5> <<3.14159:32/float>>.
<<64,73,15,208>>
6> <<3.14159:64/float>>.
<<64,9,33,249,240,27,134,110>>
```

Bit Syntax examples

- 32 bit words are a pain to unpack
- `X = 0xafab1234`
- extract 3 bits then 6 bits then 2 bits

```
<<X:3,Y:6,Z:2, ...>> = Var
```

- (`1010 1111 1010 1011 0001 0010 0011 0100`)
- 101 011111 01 = 5, 31, 1

```
1> X = 16#afab1234.
2947224116.
2> B = <<X:32/unsigned-integer>>.
<<175,171,18,52>>
3> <<P:3,Q:5,R:1,_/bits>> = B.
<<175,171,18,52>>
4> P.
5
5> Q.
15
6> R.
```

Unpacking an IPv4 Datagram

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

Binary BIFS

- `list_to_binary(ListOrDeepList) -> Binary.`
- `binary_to_list(Binary) -> List.`
- `term_to_binary(Term) -> Binary.`
- `binary_to_term(Term) -> Binary.`

  And many more
- `erl -man binary.`
- `term_to_binary` and its inverse are incredibly useful.

The awesomeness of term_to_binary

- Universal serial/deserialiaze any Erlang term.
- Used in distributed Erlang.
- Used in databases.
- Very Fast.

Example of some awesomeness

```
encode(Term, Password, PublicKey) ->
   Bin = term_to_binary(Term),
   SecretBinary = encode(Bin, Password),
   Term1 = sign(SecretBinary, PublicKey),
   Bin1 = term_to_binary(Term1),
   Len = size(Bin),
   <<Len:32/big, Bin>>.

decode(Bin, Password, PrivateKey) ->
   <<Len:32/big, B1>> = Bin,
   Term1 = binary_to_term(B1),
   Bin1 = term_to_binary(Term),
   SecretBinary = decode(Bin1, PrivateKey),
   Bin = decode(SecretBinary, Password),
   binary_to_term(Bin).
```

Guards

- Used to extend pattern matching:

```
func(X, Y) when length(X) > length(Y) ->
  ...
func(X) when is_tuple(X), size(X) > 3 ->
  ...
```

- or in `if` exressions.

- Cannot be user defined.

- Cannot have side effects.

Rest of Sequential Erlang

This is chapter 8 in the book. Mainly for reference:

- Comments.
- Block expressions.
- Escape Sequences.
- Include Files.
- Underscore variables.
- Tuple Modules.
- get/put.
- etc...

W5/F5 - Sequential Erlang 2(b)

- Chapters 9,10
- Types - Informative not mandatory.
- Dynamic and static types - Pros and cons.
- Type Inference vs. Type checking.
- Type notation.
- Dialyzer - a program to analyse for type errors.
- Compiling.
- Makefiles.
- Environment tweaking.

Types Checked at runtime.

- Erlang is dynamically typed. It makes programs easy to write but not so easy to read.
- It we see integers or string in a program what do they mean?
- Can we detect type errors at compile time?
- Can we use type information to produce optimised code?

Type Inference vs. Type checking

```
foo(X, Y) -> X + Y.


+spec bar(integer(), float()) -> float().
bar(X, Y) -> X + Y
```

In the definition of `foo` we are given no type information but we can *infer* from the usage patterns of `X` and `Y` are of type `number()`. So we could infer that the program fragment `foo("hello", 12)` was erroneous.

With type inference we detect a collision between the usage and the function definition, but we do not know which is correct.

In the definition of `bar` we say what the types are so we can check everything (both the definition and the usage) against the specification.

Dynamic and Static types

- Dynamic types - there are no type declarations, types are checked at run time.

- Static types - there are type declarations types are checked at run time.

- Both have the advantages and disadvantages.

- There is a lot of discussion about this.

Type notation

- There are primitive types, and type constructors.
- Primitive types reflect the primitive types in Erlang.
  - `integer(),atom()` ...
- Constructed types are made in three ways:
  - `{Type1(), Type2() , ...}` – tuple type
  - `Type1() | Type2() | ....` – variant type
  - `[Type()]` – list type
- Types can be annotated with a name (next slide)

5 birthday declarations

```
birthday1(N) -> N.

-spec birthday2(integer()) -> integer().
birthday2(N) -> N+1.

-spec birthday3(Years::integer()) -> Years::integer().
birthday3(N) -> N+1.

-type age() :: integer().
-spec birthday4(age()) -> age().
birthday4(N) -> N+1.

-type age1() :: Years :: integer().
-spec birthday5(age1()) -> age1().
birthday5(N) -> N+1.
```

Running typer

```
-export([add/2, bug/1, fac/1]).

add(X, Y) -> X+Y.
bug(X) -> add("hello", X).

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

```
$ typer math1_bug.erl
%% File: "math1_bug.erl"
%% ---------------------
-spec add(number(),number()) -> number().
-spec bug(_) -> none().
-spec fac(non_neg_integer()) -> pos_integer().
```

The dreaded error message

```
joearms$ typer math1_bug.erl
typer: Dialyzer’s PLT is missing
or is not up-to-date;
please (re)create it
```

Go make a cup of coffee

```
$ dialyzer --build_plt --apps erts kernel stdlib
  Compiling some key modules to native code...
    done in 1m19.68s
  Creating PLT /Users/joearms/.dialyzer_plt ...
Unknown functions:
  compile:file/2
  compile:forms/2
  compile:noenv_forms/2
  compile:output_generated/1
  crypto:des3_cbc_decrypt/5
  crypto:start/0
Unknown types:
  compile:option/0
 done in 2m18.88s
done (passed successfully)
```

Dialyzer – Descrepency analyser for Erlang programs

- Finds things in your program that will cause type errors at run-time.

- Uses “success” typing.

Working with types

- Think about your types before you write the program.
- Write functions one at a time an run the dialyzer after each you've written each new function.
- Avoid `export_all`.
- Add guards, this will improve the accuracy of type inference.

Dialyzer output

```
$ dialyzer math1_bug.erl
dialyzer math1_bug.erl
  Checking whether the PLT /Users/joe/.dialyzer_plt
  is up-to-date... yes
  Proceeding with analysis...
math1_bug.erl:7: Function bug/1 has no local return
math1_bug.erl:8: The call
   math1_bug:add("hello",X::any())
   will never return since it differs
   in the 1st argument
   from the success typing
   arguments: (number(),number())
 done in 0m0.98s
done (warnings were emitted)
```

The startup file

- The file *hidden* file `.erlang` is consulted when you start Erlang.
- This can be in the directory where Erlang was started or your home directory.
- This file can contain any Erlang commands. They are executed when you start Erlang.

Search Paths

- Code is dynamically loaded at run time.
- `code:get_path()` tells you want the current path is.
- The first time you call `foo:` the system will search for a file called `foo.beam` using the current search path.
- use `code:add_patha(Dir)` or `code:add_pathz(Dir)` in your startup file.

Compiling your program

- In the Erlang shell (`> c(FileName).`)
- In the OS command shell (`$erlc foo.erl`).
- From a Makefile.
- from rebar.

Running Erlang from the shell

```
$ erl -pa Path1 -pa Path2 -s M F Arg1 Arg2 ...
```

- Can use noshell to stop the prompt

  ```
  $erl -noshell ...
  ```

Running Erlang from a shell script

Create a file `runme` like this:

```
#!/bin/bash
$ erl -pa Path1 -pa Path2 -s M F $1 ...
```

Then run it:

```
$ chmod u+x runme
$ runme arg1 ...
```

Running Erlang from escript

```
#!/usr/bin/env escript

main(_) ->
    ...
```

- chmod the file then you can run it as a script.