Robert Virding

Principle Language Expert
at Erlang Solutions Ltd.

Erlang Solutions Ltd.

Multi-lingual Erlang

We are great believers in:

- No language/system is good at everything
- Right tool for the job

Overview

- Properties of the BEAM
- What languages?
- Basic Tools
- “Native” languages
- “Non-native” languages
- External systems
- Which one?

What IS the BEAM?

A virtual machine to run Erlang

Properties of the Erlang system

- Lightweight, massive concurrency
- Asynchronous communication
- Process isolation
- Error handling
- Continuous evolution of the system
- Soft real-time
- Support for introspection and monitoring

These we seldom have to directly worry about in a language, except for receiving messages

Properties of the Erlang system

- Immutable data
- Pattern matching
- Functional language
- Predefined set of data types
- Modules
- No global data

These are what we mainly “see” directly in our languages

Adding “new” datatypes

- Erlang has records!
- We have to fake it
  - use existing datatype, tuple
  - special format, record name as first element
  - special syntax and libraries to create and recognise the new “type”

What languages?

- Anything written in another language
  - Config files
  - DSLs
  - Other “languages”
  - ...

Basic tools

- leex – lexical scanner generator
- yecc – parser generator
- syntax tools – for building erlang code
- XML parsers (xmerl)
- Erlang compiler (of course)

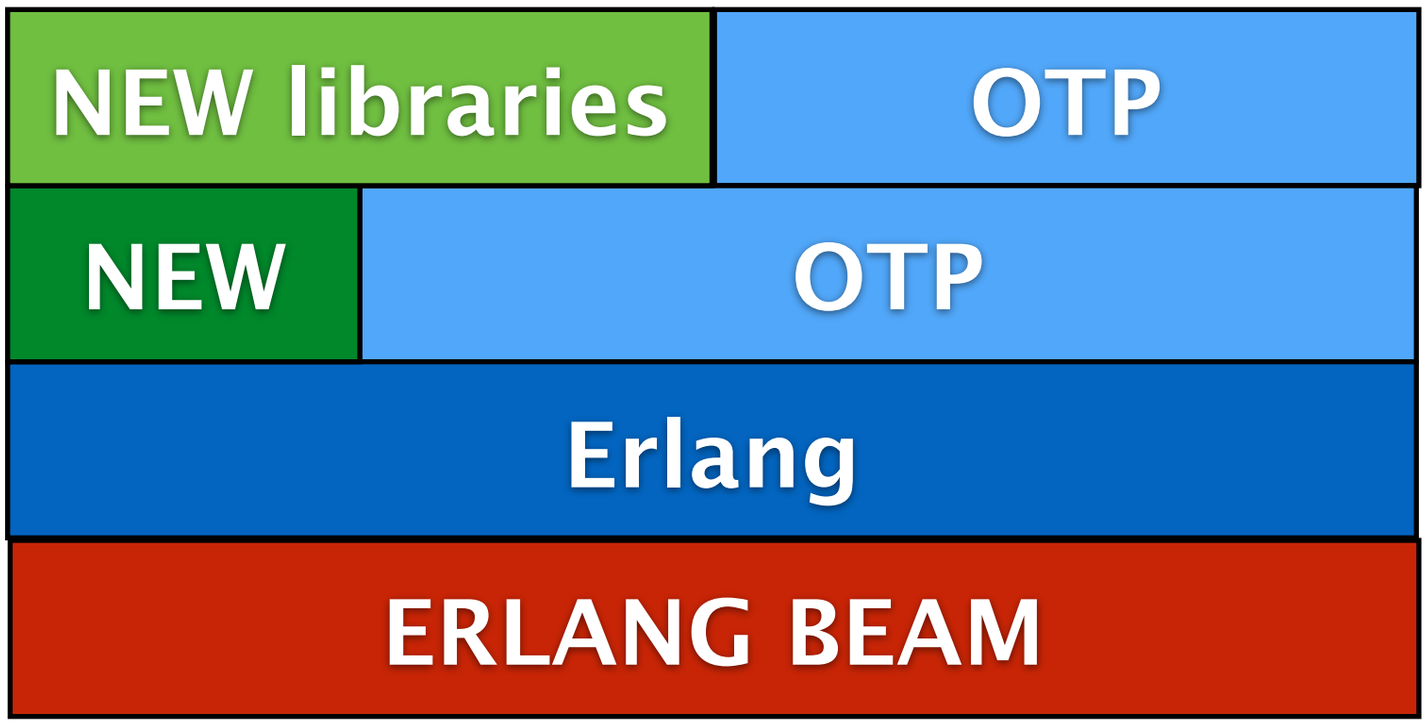
NEW SKIN FOR THE OLD CEREMONY

LEONARD COHEN

New Skin for the Old Ceremony

- Languages which keep the basic Erlang execution model and data types
  - New syntax
  - Different “packaging”
- Elixir
- LFE (Lisp Flavoured Erlang)

New Skin for the Old Ceremony



- The basic properties of these languages are based on the properties of the Erlang and the Erlang VM
- Make full use of the Erlang/OTP libraries

LFE (Lisp Flavoured Erlang)

The goal was:

- Provide lots of lisp goodies
  - real homoiconicity and macros (yay!)
- Seamlessly interact with vanilla Erlang/OTP
  - be able to freely mix vanilla code and LFE code
- Small core language
- Same speed as vanilla Erlang

LFE (Lisp Flavoured Erlang)

➡ modify the language to fit with Erlang/OTP

- no mutable data
- only have standard Erlang data types
  - Erlang style records
- Erlang style modules and functions
  - no functions with variable number of arguments
- macros are only compile-time
- Lisp-2 (instead of Lisp-1)

- is as fast as vanilla Erlang

Elixir

- “Elixir is a dynamic, functional language designed for building scalable and maintainable applications.”
- “Elixir is influenced by Ruby”
  - “Elixir is NOT Ruby on the Erlang VM”
- Elixir has meta programming capabilities by macros
  - they cannot change the syntax
- Many libraries and interfaces standardised and rewritten

Elixir - “new” datatypes

- Elixir has records but seldom used
- Elixir has structs
- We have to fake it
  - use existing datatype, map
  - special format special key ‘__struct__’ with value of struct name
  - Special syntax and libraries to create and recognise the new “type”

Non-native languages

- Languages which are not just basic Erlang
  - different semantics
  - non-Erlang datatypes
  - non-Erlang handling of data

- Erlog (prolog)
- Luerl (Lua)

Erlog

- Standard prolog, at least a strict subset
- Completely different semantics to Erlang
  - backtracking
  - logical variables
  - unification
- Good mapping between Erlog <-> Erlang data structures
  - except for logical variables

Luerl

- Implements standard Lua 5.2
- Lua is
  - Simple, rather neat little imperative language
  - Dynamic language
  - Lexically scoped
  - Mutable variables/environments/global data
  - Common scripting language in games

Luerl - Lua datatypes

- nil
- booleans
- numbers (floating point)
- strings
- mutable, global key-value tables
  - which it uses as tables/arrays/lists/kitchen sink
  - updates are visible everywhere

External systems

- Erlang ports
  - linked-in drivers
- NIFs (Natively Implemented Functions)
- C-nodes (distributed Erlang)
- UDP/TCP

Erlang ports

- Interface to the outside world
- Oldest mechanism
- Resemble processes
  - message based interface
  - error handling
- Fail-safe
- Byte streams
  - support for packets
- Support for en-/decoding erlang data structures

Erlang ports

```
/* echo.c */

#include <stdio.h>
#define BUFFER_LENGTH 80

int main() {
    char line[BUFFER_LENGTH];
    while (1) {
        if (fgets(line, BUFFER_LENGTH, stdin) != NULL) {
            printf("%s", line);
            fflush(stdout);
        }
        else {
            return 0;
        }
    }
}
```

Erlang ports

```
start(ExtPrg) ->
    open_port({spawn, ExtPrg}, [stream, {line, 80}).

echo_line(Port, Line) ->
    Port ! {self(), {command,Line}},
    get_reply(Port, []).

get_reply(Port, Acc) ->
    receive
        {Port, {data,{eol,Chars}}} ->
            {ok,lists:flatten(Acc ++ Chars)};
        {Port, {data,{noeol,Chars}}} ->
            get_reply(Port, Acc ++ Chars)
    end.
```

Erlang ports: linked-in drivers

- Move functionality inside the Erlang VM
- Behaves like a port
- Can be more efficient than “normal” ports
- No safety
  - “other end” is internal

NIFs (Natively implemented functions)

- Implement functions in C
- Call them as “normal” Erlang functions
- Large library for accessing Erlang data structures in C
- Support for threads
- No safety

NIFs (Natively implemented functions)

```
static ERL_NIF_TERM
i2c_read(ErlNifEnv* env, int argc, const ERL_NIF_TERM argv[]) {
  unsigned int fd, addr, len;
  unsigned char buf[1024], *bdata;
  ERL_NIF_TERM term;

  /* Get and parse arguments. */
  if (!enif_get_uint(env, argv[0], &fd) ||
      !enif_get_uint(env, argv[1], &addr) ||
      !enif_get_uint(env, argv[2], &len))
    return enif_make_badarg(env);

  /* Select i2c address and read data. */
  if (ioctl(fd, I2C_SLAVE, addr >> 1) == 0 &&
      read(fd, buf, len) == len) {
    bdata = enif_make_new_binary(env, len, &term);
    memcpy(bdata, buf, len);
    return term;
  }
  else
    return enif_make_badarg(env);
}
```

NIFs (Natively implemented functions)

```
loop(St) ->
    receive
        {i2c_request,From,{read,Addr,Bc}} ->    %Read bytes
            Buf = i2c_read(St#state.i2c, Addr, Bc),
            reply(From, Buf),
            loop(St);
        {i2c_request,From,{write,Addr,Buf}} ->  %Write bytes
            ok = i2c_write(St#state.i2c, Addr, Buf),
            reply(From, ok),
            loop(St);
        {i2c_request,From,{request,Addr,Req,Wait,Bc}} ->
            I2c = St#state.i2c,
            ok = i2c_write(I2c, Addr, Req),
            timer:sleep(Wait),                  %Need to wait before reading
            Buf = i2c_read(I2c, Addr, Bc),
            reply(From, Buf),
            loop(St);
        {i2c_request,From,{requests,Reqs}} ->   %Do requests
            Reps = do_requests(Reqs, St#state.i2c),
            reply(From, Reps),
            loop(St);
        stop ->
            ok = i2c_close(St#state.i2c)
    end.
```

C-nodes

- External OS process behaves as a distributed Erlang node
- Fail safe
  - other end a node
- “Natural” message interface from Erlang

C-nodes

https://github.com/rtraschke/erlang-lua

Which one?: native languages

+ Complete access to Erlang/VM properties
+ Fastest on the Erlang VM
+ Conceptually easy to interface (just Erlang)
+ Safe

- Only Erlang equivalent languages
- Usually “need” large environment to be usable
  - Libraries, tools, REPL, Emacs mode, …

Which one?: non-native languages

+ Good access to Erlang/VM properties
+ Fast data transfer with Erlang
+ Other languages
+ Safe

- Slower than “normal” implementation
- Can be costly to implement
- Need an environment for non-Erlang features
  - Global state, mutable data, …

Which one?: external systems

+ Can access existing languages/systems
+ Can monitor and manage systems
+ Maximum efficiency available

- Generally slower interface
- Safe or unsafe
  - External is safe
  - Internal is unsafe

Thank you

Robert Virding: robert.virding@erlang-solutions.com

@rvirding