Robert Virding

Principle Language Expert
at Erlang Solutions Ltd.

LFE - a lisp on the Erlang VM

LFE - Lisp Flavoured Erlang

What LFE isn't

- It isn't an implementation of Scheme
- It isn't an implementation of Common Lisp
- It isn't an implementation of Clojure

- Properties of the Erlang VM make these languages difficult to implement *efficiently*

What LFE is

- LFE is a proper lisp based on the features and limitations of the Erlang VM
- LFE coexists seamlessly with vanilla Erlang and OTP
- Runs on the standard Erlang VM

Overview

- A little history
- A bit of philosophy and a rationale
- The goal
- What is the BEAM?
- Properties of the BEAM/LFE
- Implementation

The problem

- Ericsson's "best seller" AXE telephone exchanges (switches) required large effort to develop and maintain software.

- The problem to solve was how to make programming these types of applications easier, but keeping the same characteristics.

Problem domain

- Lightweight, massive concurrency
- Fault-tolerance must be provided
- Timing constraints
- Continuous maintenance/evolution of the system
- Distributed systems

Properties of the Erlang system

- Lightweight, massive concurrency
- Asynchronous communication
- Process isolation
- Error handling
- Continuous evolution of the system
- Soft real-time

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

Properties of the Erlang system

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

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

Some reflections

We were **NOT** trying to implement a functional language

We were **NOT** trying to implement the actor model

WE WERE TRYING TO SOLVE THE PROBLEM!

Some reflections

- This made the development of the language/system very focused
- We had a clear set of criteria for what should go into the language/system
  - Was it useful?
  - Did it or did it not help build systems?

The language/system evolved to solve the problem

The LFE goal

A “proper” lisp

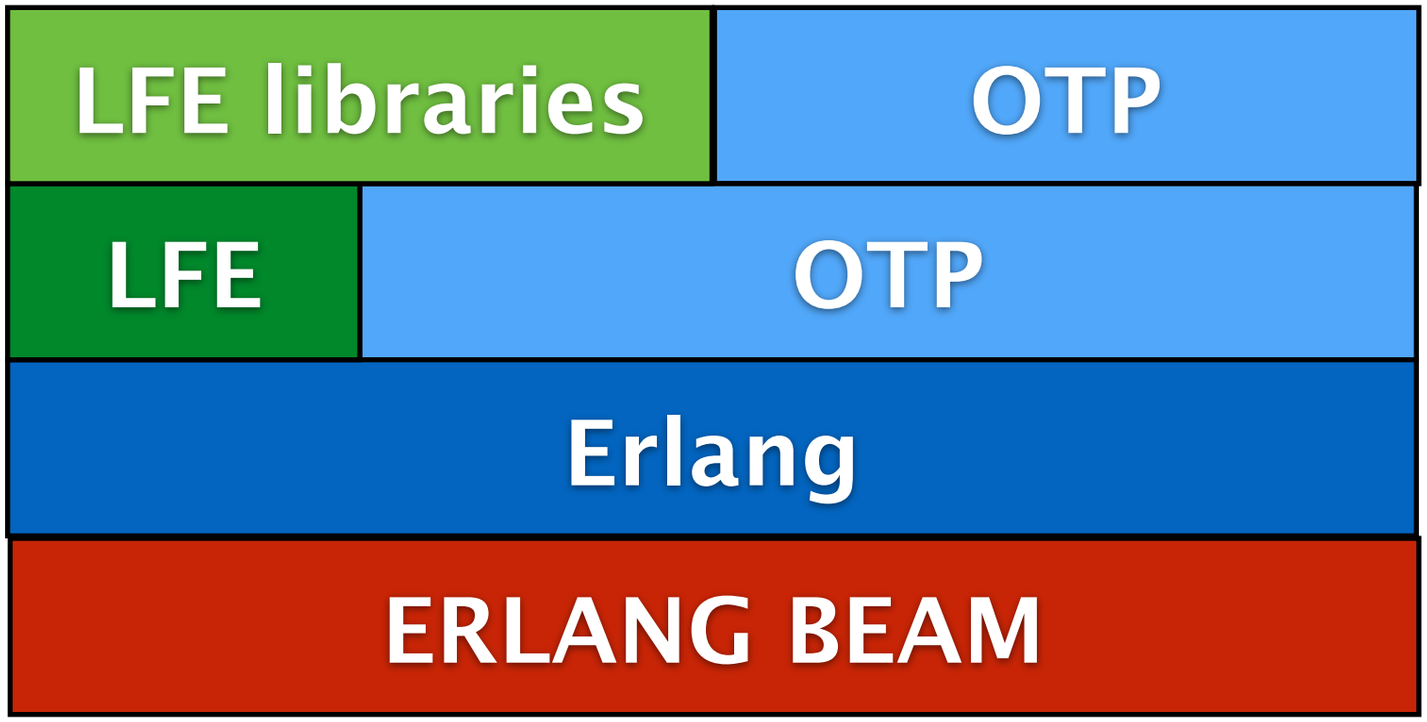
Efficient implementation on the BEAM

Seamless interaction with Erlang/OTP and all libraries

NEW SKIN FOR THE OLD CEREMONY

LEONARD COHEN

New Skin for the Old Ceremony



The thickness of the skin affects how efficiently the new language can be implemented and how seamlessly it can interact

What IS the BEAM?

A virtual machine to run Erlang

Properties of the BEAM

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

Features of LFE

- Syntax
- Data types
- Modules/functions
- Lisp-1 vs. Lisp-2
- Pattern matching
- Macros

Syntax

- `[ … ]` an alternative to `( … )`
- Symbol is any number which is not a number
  - `| a quoted symbol |`
- `() [] {} . ‘ ` , ,@ #(#b(#m(` separators

- `#( … )` tuple constant
- `#b( … )` binary constant
- “abc” <—> `(97 98 99)`
- `#\a` or `#\xab;` characters

Data types

- LFE has a fixed set of data types
  - Numbers
  - Atoms (lisp symbols)
  - Lists
  - Tuples (lisp vectors)
  - Maps
  - Binaries
  - Opaque types

Atom/symbols

- Only has a name, no other properties
- **ONE** name space
- No CL packages
  - No name munging to fake it
  - ~~**foo** in package **bar** => **bar:foo**~~
- Booleans are atoms, **`true`** and **`false`**

Binaries

```
(binary 1 2 3)
(binary (t little-endian (size 16))
        (u (size 4)) (v (size 4))
        (f float (size 32))
        (b bitstring))
```

- Byte/bit data with constructors
- Properties are type, size endianess, sign

- But must do `((foo a 35))`

Binaries

```
(binary (ip-version (size 4)) (h-len (size 4))
        (srvc-type (size 8)) (tot-len (size 16))
        (id (size 16)) (flags (size 3))
        (frag-off (size 13)) (ttl (size 8))
        (proto (size 8)) (hrd-chksum (size 16))
        (src-ip (size 32)) (dst-ip (size 32))
        (rest bytes))
```

- IP packet header

Modules and functions

- Modules are very basic
  - Only have name and exported functions
  - Only contains functions
  - Flat module space
- Modules are the unit of code handling
  - compilation, loading, deleting
- Functions only exist in modules
  - Except in the shell (REPL)
- **NO** interdependencies between modules

Modules and functions

```
(defmodule arith
  (export (add 2) (add 3) (sub 2)))

(defun add (a b) (+ a b))

(defun add (a b c) (+ a b c))

(defun sub (a b) (- a b))
```

- Function definition resembles CL
- Functions CANNOT have a variable number of arguments!
- Can have functions with the same name and different number of arguments (arity), they are different functions

Modules and functions

- LFE modules can consist of
  - Declarations
  - Function definitions
  - Macro definitions
  - Compile time function definitions
- Macros can be defined anywhere, but must be defined before being used

Lisp-1 vs. Lisp-2

- How symbols are evaluated in the function position and argument position
- In Lisp-1 symbols only have value cells

```
(foo 42 bar)
```

value

- In Lisp-2 symbols have value and function cells

```
(foo 42 bar)
```

function value

Lisp-1 vs. Lisp-2

```
(defun foo (x y) …)
(defun foo (x y z) …)

(defun bar (a b c)
  (let ((baz (lambda (m) …)))
    (baz c)
    (foo a b)
    (foo 42 a b)))
```

- With Lisp-1 in LFE I can have multiple top-level functions with the same name, `foo/2` and `foo/3`
- But only one local function with a name, baz/1

THIS IS INCONSISTENT!

Lisp-1 vs. Lisp-2

```
(defun foo (x y) …)
(defun foo (x y z) …)

(defun bar (a b c)
  (flet ((baz (m) …)
         (baz (m n) …))
    (foo a b)
    (foo 42 a b)
    (baz c)
    (baz a c)))
```

- With Lisp-2 in LFE I can have multiple top-level *and* local functions with the same name, `foo/2`, `foo/3` and `baz/1`, `baz/2`

THIS IS CONSISTENT!

Lisp-1 vs. Lisp-2

- Erlang/LFE functions have both name and arity
- Lisp-2 fits Erlang VM better
- LFE is Lisp-2, or rather Lisp-2+

Pattern matching

- Pattern matching is a BIG WIN™
- The Erlang VM directly supports pattern matching
- We use pattern matching everywhere
  - Function clauses
  - `let`, `case` and `receive`
  - In macros `cond`, `lc` and `bc`

Pattern matching

```
(let ((<pattern> <expression>)
      (<pattern> <expression>)
  …)

(case <expression>
  (<pattern> <expression> …)
  …)

(receive
  (<pattern> <expression> …)
  (<pattern> <expression> …)
  …)
```

- Variables are only bound through pattern matching

Pattern matching

```
(defun name
  ([<pat1> <pat2> …] <expression> …)
  ([<pat1> <pat2> …] <expression> …)
  …)

(cond (<test> …)
      ((?= <pattern> <expr>) …)
      …)
```

- Function clauses use pattern matching to select clause

Pattern matching

```
(defun ackermann
  ([0 n] (+ n 1))
  ([m 0] (ackermann (- m 1) 1))
  ([m n] (ackermann (- m 1) (ackermann m (- n 1)))))

(defun member (x es)
  (cond ((=:= es ()) 'false)
        ((=:= x (car es)) 'true)
        (else (member x (cdr es)))))

(defun member
  ([x (cons e es)] (when (=:= x e)) 'true)
  ([x (cons e es)] (member x es))
  ([x ()] 'false))
```

Macros

- Macros are UNHYGIENIC
- No (gensym)
  - Cannot create unique atoms
  - Unsafe in long-lived systems
- Only compile-time at the moment
  - Except in the shell (REPL)
- Core forms can never be shadowed

Macros

```
(defmacro add-them (a b) `(+ ,a ,b))

(defmacro avg args                          ;(&rest args) in CL
  `(/ (+ ,@args) ,(length args)))

(defmacro list*
  ((list e) e)
  ((cons e es) `(cons ,e (list* . ,es)))
  (() ()))
```

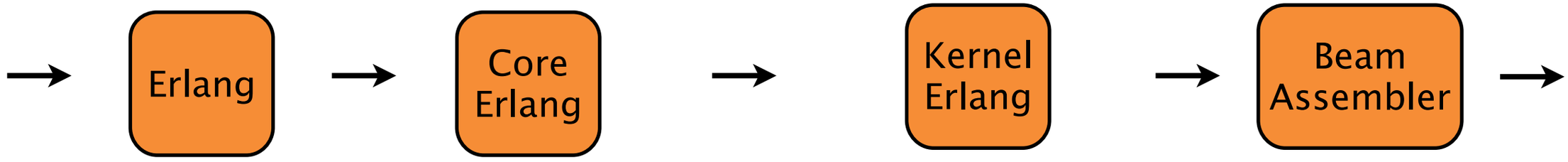
- Macros can have any other number of arguments
  - But only macro definition per name
- Macros can have multiple clauses like functions
  - The argument is then the list of arguments to the macro
- We have the backquote macro

Implementation: Erlang compiler

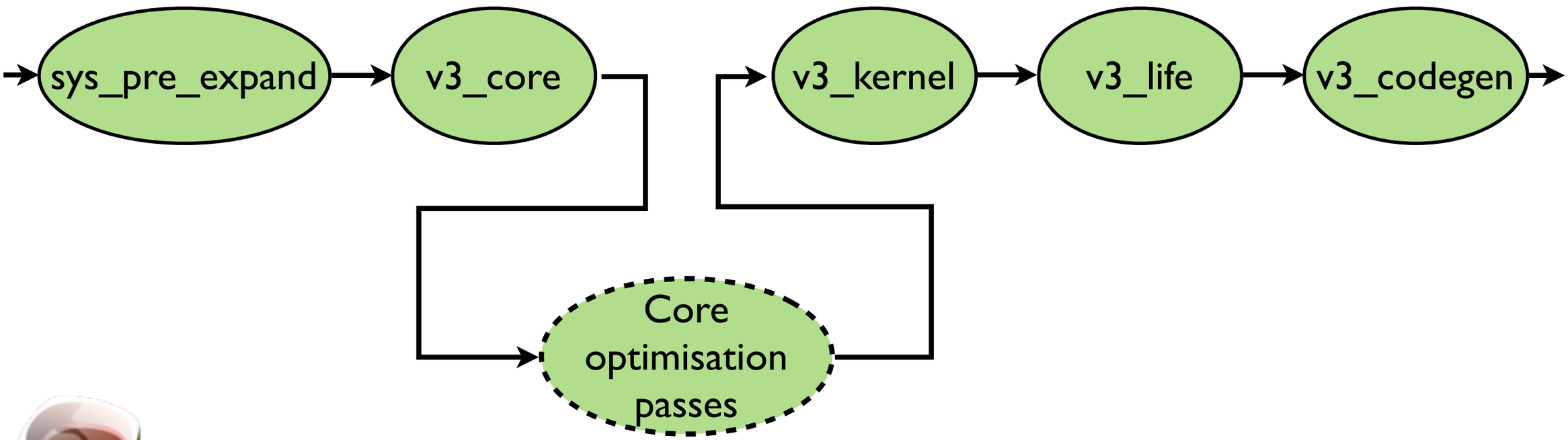
- Can work on files and Erlang abstract code
- Can generate .beam files or binaries
- Has Core, a nice intermediate language
  - Can be input to the compiler
  - simple and regular
  - easier to compile to

Implementation: Erlang compiler

Internal languages



Internal modules



Implementation: Core erlang

- Simple functional language
- "normal" lexical scoping
- Has just the basics
  - no records
  - no list comprehensions
- Supports pattern matching (yeah!)
- Most optimisations done on core
- Dialyzer speaks Core
  - sort of :-(

Implementation: Core erlang

```
(defun sum
  ([(cons h t)] (+ h (sum t)))              ;`(,h . ,t)
  ([()] 0))

'sum'/1 =
    fun (_cor0) ->
        case _cor0 of
      <[H|T]> when 'true' ->
          let <_cor1> =
              apply 'sum'/1(T)
          in  call 'erlang':'+'(H, _cor1)
      <[]> when 'true' -> 0
      ( <_cor2> when 'true' ->
            ( primop 'match_fail' ({'function_clause',_cor2})
              -| [{'function_name',{'sum',1}}] )
        -| ['compiler_generated'] )
    end
```

Implementation: Core LFE

```
(case expr clause ...)
(if test true false)
(receive clause ... (after timeout ...))
(catch ...)
(try expr (case ...) (catch ...) (after ...))
(lambda ...)
(match-lambda clause ...)
(let ...)
(let-function ...), (letrec-function ...)
(cons h t), (list ...) (tuple ...) (binary ...)
(func arg ...), (funcall var arg ...)
(call mod func arg ...)
(define-function name lambda|match-lambda)
(define-macro name lambda|match-lambda)
```

WHY? WHY? WHY?

I like Lisp

I like Erlang

I like to implement languages

So doing LFE seemed natural

Robert Virding: rvirding@gmail.com @rvirding

LFE

http://lfe.io/

https://github.com/rvirding/lfe

https://github.com/lfe

http://groups.google.se/group/lisp-flavoured-erlang

#erlang-lisp @ErlangLisp