Erlang SOLUTIONS

www.erlang-solutions.com

LFE - a real lisp in the Erlang ecosystem

Robert Virding

The LFE goal

A "proper" lisp

Efficient implementation on the BEAM

Seamless interaction with Erlang/OTP and all libraries.

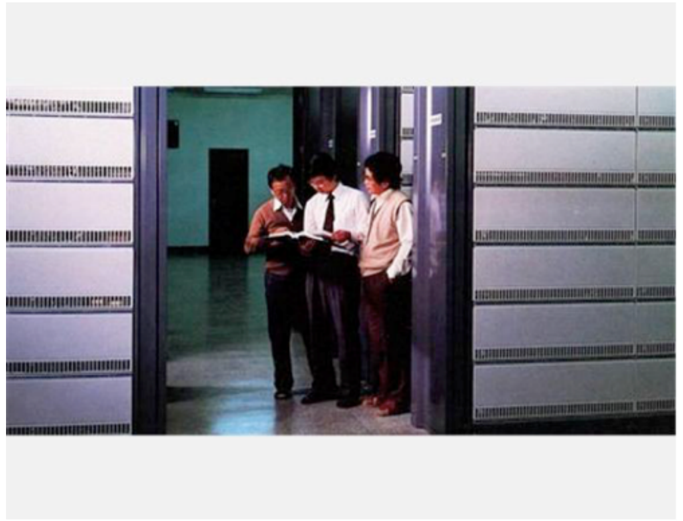
Overview

- Background
- Erlang Ecosystem
- LFE

Background: 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.



Background: some reflections

We were **not** out to implement a functional language

We were **not** out to implement the actor model

WE WERE TRYING TO SOLVE THE PROBLEM

Background: problem domain

- Handle a very large numbers of concurrent activities.
- Actions must be performed at a certain point in time or within a certain time.
- System distributed over several computers.
- Interaction with hardware.
- Very large software systems.
- Complex functionality such as feature interaction.
- Continuous operation over many years.
- Software maintenance (reconfiguration etc.) without stopping the system.
- Stringent quality and reliability requirements.
- Fault tolerance both to hardware failures and software errors.

Not just telecom

Bjarne Däcker, November 2000 – Licentiate Thesis

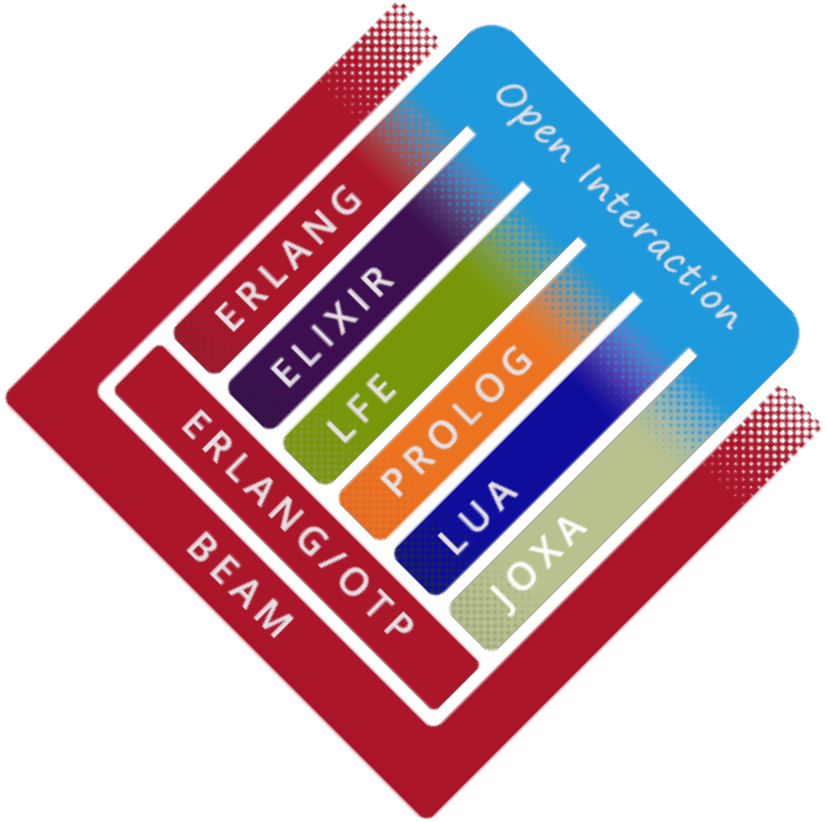
Background

Erlang and the system around it was designed to solve this type of problem

Erlang/OTP provides direct support for these issues

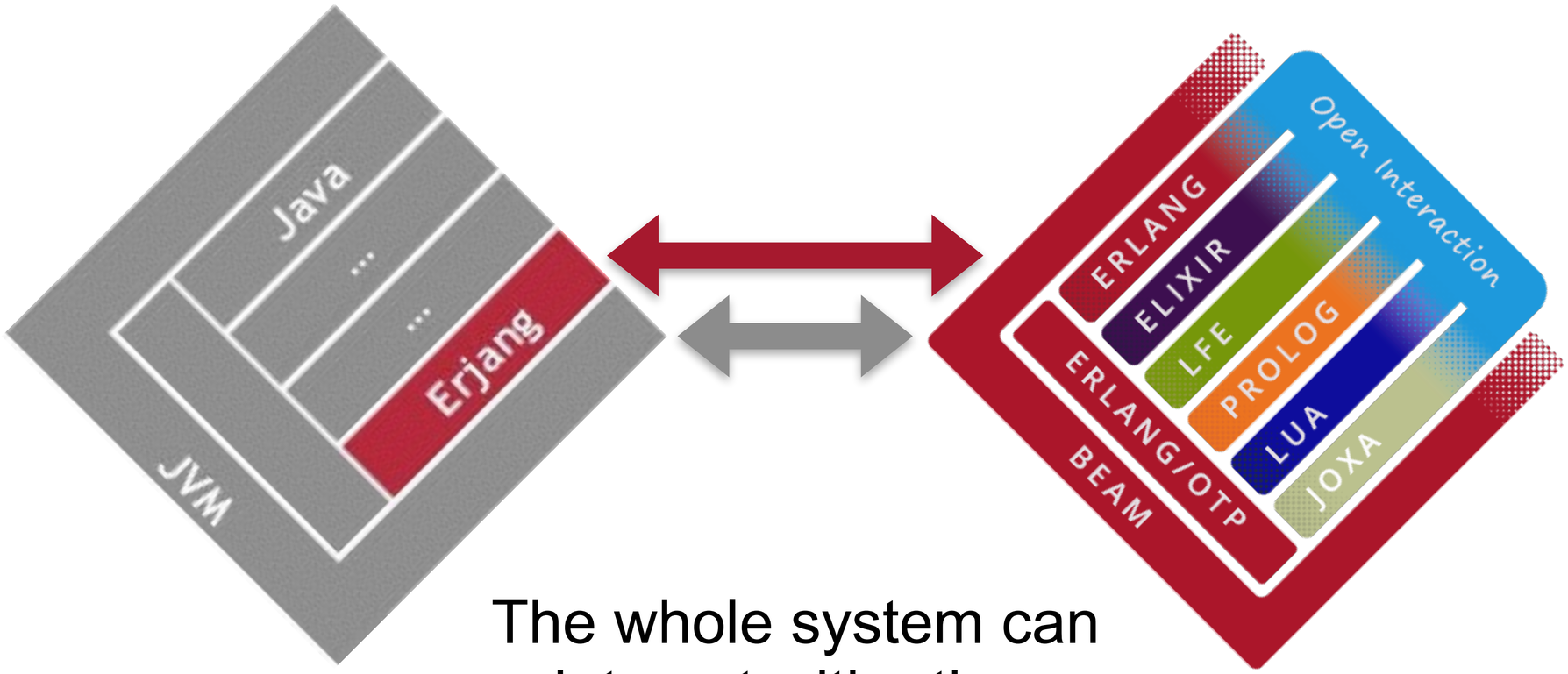
Erlang Ecosystem

Languages built/running on top of the BEAM, Erlang and OTP.



By following "the rules" the languages openly interact with each other

Erlang Ecosystem



The whole system can interact with other systems

What is the BEAM?

A virtual machine to run Erlang

Properties of the BEAM

- Lightweight, massive concurrency
- Asynchronous communication
- Process isolation
- Error handling
- Continuous evolution of the system
- Soft real-time
- Transparent SMP/multi-core support

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

Properties of the BEAM

- 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

Why Lisp?

- Do we really something so old?

```
DEFINE ((
(MEMBER (LAMBDA (A X) (COND ((NULL X) F)
        ( (EQ A (CAR X) ) T) (T (MEMBER A (CDR X))) )))
(UNION (LAMBDA (X Y) (COND ((NULL X) Y) ((MEMBER
        (CAR X) Y) (UNION (CDR X) Y)) (T (CONS (CAR X)
        (UNION (CDR X) Y))) )))
(INTERSECTION (LAMBDA (X Y) (COND ((NULL X) NIL)
        ( (MEMBER (CAR X) Y) (CONS (CAR X) (INTERSECTION
        (CDR X) Y))) (T (INTERSECTION (CDR X) Y)) )))
))
INTERSECTION ((A1 A2 A3) (A1 A3 A5))
UNION ((X Y Z) (U V W X))
```

Why Lisp?

- Do we really want something so old?
- Fortunately we don't have to

```
(defun union
  ((() set) set)
  (((cons x xs) set)
   (if (lists:member x set) (union xs set)
     (cons x (union xs set)))))

(defun intersection
  ((() _) ())
  (((cons x xs) set)
   (if (lists:member x set) (cons x (intersection xs set))
     (intersection xs set))))
```

Why Lisp?


```
1 56.0 9

bert more-of do if size >

(1 2 3)
(a b c)
(a b (x 1 y) 3)
(> size 4)
(if (> size 4)
  (bump-it)
  (drop-it))

(defun test (size)
  (if (> size 4)
    (bump-it)
    (drop-it)))
```

- Numbers
- Symbols
- Lists
- Lists, hmm ...
- Lists, but this looks like code
- But code is lists

Why Lisp?

- A lot has changed since 1958 ... even for Lisp: it has now even more to offer
- It's a programmable programming language
- As such, it's an excellent language for exploratory programming
- Many are drawn to the beauty of the near syntaxlessness of the language
- Due to its venerable age there is an enormous body of code to draw from

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
- Runs on the standard Erlang VM
- LFE coexists seamlessly with OTP and the other languages in the Erlang ecosystem

Features of LFE

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

Data types

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

Atoms/symbols

- Only has a name, no other properties
- ONE name space

- No CL packages or namespaces

- ~~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

Binaries


```
(binary (ip-version (size 4)) (hdr-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)) (hdr-chksum (size 16))
        (src-ip (size 32)) (dst-ip (size 32))
        (rest bytes))
```

- IPv4 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
- Support for multiple modules in one file

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 names and different number of arguments (arity), they are different functions

Modules and functions

- LFE modules can consist of
  - Attributes
  - Metadata
  - 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> …)
  (<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
  - But not so bad as all variables are scoped and cannot be changed
- 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 one 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

Code example


```
(defun ringing-a-side (addr b-pid b-addr)
  (receive
    ('on-hook
     (! b-pid 'cleared)
     (tele-os:stop-tone addr)
     (idle addr))
    ('answered
     (tele-os:stop-tone addr)
     (tele-os:connect addr b-addr)
     (speech addr b-pid b-addr))
    (`#(seize ,pid)
     (! pid 'rejected)
     (ringing-a-side addr b-pid b-addr))
    (_
     (ringing-a-side addr b-pid b-addr))
  ))
```

```
(defun ringing-b-side (addr a-pid)
  (receive
    ('cleared
     (tele-os:stop-ring addr)
     (idle addr))
    ('off_hook
     (tele-os:stop-ring addr)
     (! a-pid 'answered)
     (speech addr a-pid 'not-used))
    (`#(seize ,pid)
     (! pid 'rejected)
     (ringing-b-side addr a-pid))
    (_
     (ringing-b-side addr b-pid))))
```

Ongoing work

- Call inter-module macros (mod:macro …)
  - Compile-time so far, run-time sort of (but is it used?)
- Adding type notations
- Lisp Machine Flavors
  - Pre-cursor to CLOS
  - A not too-bad mapping with many cool properties
- Clojure interface
- Lisp Machine Structs
  - More versatile formatting and access
  - Subsumes records and Elixir structs

WHY? WHY? WHY?

I like Lisp

I like Erlang

I like to implement languages

So implementing LFE seemed natural

Robert Virding
rvirding@gmail.com
robert.virding@erlang-solutions.com
@rvirding

LFE
http://lfe.io/
https://github.com/rvirding.lfe
https://groups.google.se/group/lisp-flavoured-erlang
Slack: https://lfe.slack.com/
IRC: #erlang-lisp
Twitter: @ErlangLisp