Lisp Flavoured Erlang
LFE

Adding a new flavour to Lisp

Robert Virding

What LFE isn't

- It isn't an implementation of Scheme
- It isn't an implementation of Common Lisp
  - In fact neither are possible on the Erlang VM
    (Global data, destructive operations, ...)
- It isn't an implementation of Clojure
  - Not really useful here
    (No Java classes, wrong concurrency model, ...)

What LFE is

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

- Runs on the standard Erlang VM

Erlang – problem domain

First principles for language properties

- Lightweight concurrency
- Asynchronous communication
- Error handling
- Process isolation
- Continuous evolution of system
- High-level language

Erlang – influencing factors

- No global data
- No mutable data
- Standard Erlang data types
- Pattern matching & guards
- Erlang functions
- Erlang modules
- Compiler/interpreter

Data types

Erlang has fixed set of data types

- Numbers – integers & floating point
- Atoms (lisp symbols)
- Lists
- Tuples (lisp vectors)
- Binaries
- Opaque types

Atoms/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

- Byte/bit data with constructors

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

- Properties – type, size, endianess, sign, unit
- Known types – integers, floats, binaries, unicode characters

Binaries

- IP packet header

```
(binary (ip-version (size 4)) (h-len (size 4))
        (s-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))
```

- But must do `((foo a 35))` ☹

Records

- Do NOT define new data types
- Records are tuples
- Compile time only! ☹
- Provide named elements to a tuple
- Tags tuple with record name

```
#(person “Robert Virding”
         56
         (hacker erlang lisp))
```

Records

```
(defrecord name field-def-1
                field-def-2 ...)

field-def = name | (name default-value)

➔ (make-name field-name val ...)
  (is-name rec)
  (match-name field-name pat ...)
  (name-field rec)
  (set-name-field rec val)
  (set-name rec field-name val ...)
  ...
```

Syntax

`[ ... ]` an alternative to `( ... )` (Scheme)

Symbol is any atom which is not a number

  `|quoted symbol|`

`( ) [ ] { } . ´ ` , ,@ #(#b(` separators

`#( ... )` tuple constant

`#b( ... )` binary constant

`"abc"` ⇔ `(95 96 97)` needs quoting ☹

`#\a` or `#\xab;` characters

Pattern matching

- Pattern matching is a BIG WIN
- Erlang VM supports pattern matching

➔ We use pattern matching everywhere

Function clauses, **`let`**, **`case`** and **`receive`**

Macros **`cond`**, **`lc`**, **`bc`**

Almost a nice as in vanilla Erlang

Pattern matching

- Variables are only bound through pattern matching

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

(case <expression>
  (<pattern> <expression> ...)
  (<pattern> <expression> ...)
  ...)
```

Pattern matching

- Receive

```
(receive
  (<pattern> ...)
  (<pattern> ...)
  ...
  (after timeout
    ...))
```

- Cond

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

Patterns

- Like in vanilla Erlang patterns look like constructors

  -
    ```
    (binary (f float (size 32))
            (b bitstring))
    ```
- Use quote `'` for literal values
  - `(tuple 'ok val)`
- Ambivalent for lists ☹
  - `(list a b c)    and (a b c)`
  - `(cons h t)      and (h . t)`

Patterns

- Have aliases
  - `(= (tuple 'ok val) ret)`
  - Checked in lint
- Anonymous variable
  - `_`
- Multiply occurring variables not allowed
  - No implicit equality test
- Tuple and binary constants match literally

Guards

```
(when (> x 5) (is_list y))
```

- Guards are `(when <test> ...)` expressions directly after the pattern
- Guards are simple tests with only predefined operators (like in vanilla Erlang)
- Need to use explicit equality tests

```
(tuple x x1) (when (=:= x x1))
```

- Can be used after any pattern!

The Ugly

- Calling local functions and known BIFs is like “normal” lisp:

```
(! pid (tuple (self) (get-value x y z)))
```

- General call to function in another module:

```
(call other-mod other-func x y z)
```

- Usual call to known module and function:

```
(: lists member 'allan names) ☹
```

- Users would like something like:

```
(lists:member 'allan names)
```

Functions

- Erlang functions have both name and arity (no of arguments)
- So `foo/0` and `foo/1` are different functions
- Each Erlang function has only fixed number of arguments

➔ LFE must do the same

Functions

- So typically

```
(defun start (what)
  (foo what ()))       ;Default options

(defun start (what opts)
  ...)
```

Function heads

- Can do

```
(defun foo (a b c)
  (case a
    ((tuple 'ok val)
     (case b
       ((h . t)
        (case h
          ...))
       (() ...)))
    ...))
```

Function heads

- But clearer

```
(defun foo
  ([(tuple 'ok val) (h . t) c]
   ...)
  ([(tuple 'ok val) () c]
   ...))
```

- Pattern matching compiler handles this very efficiently!

Function definition

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


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

- Uses pattern matching and will be more efficient

Lisp-1 vs. Lisp-2

- Tried Lisp-1 but it didn't really work, resulted in funny behaviour
- Erlang functions have name *and* arity
- Lisp-2 fits Erlang VM better
- So we have Lisp-2, or rather Lisp-2+
- Result is more consistent and better (I think)

Lisp-1 vs. Lisp-2

- In Lisp-1

```
(define (foo x y) ...)
(define (bar x y)
  (let ((foo (lambda (a) ...)
            ...)
    (foo x y)
    ...)
```

- Which `foo` should you call?
  - Local `foo` variable and get a `bad_arity` error?
  - Global `foo/2` and succeed?

Lisp-1 vs. Lisp-2

- We follow CL here:

```
(defun foo (x) ...)
(defun foo (x y) ...)

(defun bar (x y z)
  (flet ((foo (x y) ...))     ;Shadows top
    (let ((bar (lambda (x) ...))
      (foo (funcall bar x) y)
      (foo z)                 ;Calls top
      ...)))
```

- Also have `(fletrec ((foo (x y) ...)) ...)`

Macros

- Macros are UNHYGIENIC!
- No **`(gensym)`**
  - Unsafe in long-lived systems
  - But probably must have
- Really only compile time at the moment
  - Except in interpreter and shell
- Core forms can ***never*** be shadowed

Macros

- CL based but with pattern matching

```
(defmacro foo (a b) ...)
(defmacro bar
  (pat ...)
  (pat ...))
```

- Pattern matches whole argument list
- Backquote `macro
- Scheme R5RS based syntax rules with ellipses

Erlang Modules

- Must use the existing module system:
  - Very basic – flat module space
  - All functions exist in modules
  - Functions only exist in modules
  - Modules are the unit of compilation
  - Modules are the unit of code handling
  - Modules really only have name and exported functions
  - There are NO interdependencies between modules!

LFE Modules

- LFE module consists of
  - Macro definitions
  - Module definition
  - Macro calls
  - Function definitions
  - Compile time function definitions
- Macros can be defined anywhere but must be defined before being used
- Macros can define macros and functions

LFE Modules

```
(defmodule foo
  (export (start 0) (start 1)
          (stop 0)
          (call 2) (cast 2))
  ;; Only save using module name.
  (import (from lists
                (all 2) (map 2)))
  (author "Robert Virding")
  ...)
```

- Module definition must be first non-macro form

Function scoping

- Within a module:
  - Default predefined Erlang BIFs
  - Explicit imports
  - Top functions in module
  - Local functions defined by flet, fletrec
- So no problem redefining Erlang BIFs or imports. Macros!
- Core forms can ***never*** be shadowed

Core Forms

```
(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)
```

Core macros

```
(: mod func arg ...)                 ;Literal mod name
(flet ...), (fletrec ...)
(let* ...), (flet* ...)
(cond ...)                           ;(?= pat expr)
(andalso ...), (orelse ...)
(do ...)                             ;Scheme
(lc (qual ...) expr ...)             ;(<- pat expr)
(bc (qual ...) expr ...)             ;(<= pat bin)
(fun name arity), (fun mod name arity)
(++ ...)
```

- And a bunch of CL inspired macros - **`defun`**, **`defmacro`**

LFE compiler

- 3 passes
  - Macro expansion
  - Linting (error checking)
  - Code generation
- Lint and codegen only see LFE core forms
- Generates Core erlang
- LFE core forms ⇔ Core erlang
  - So compiler relatively simple

LFE interpreter

- Can evaluate all LFE expressions

BUT ☹

- Erlang VM only supports compiled modules and functions!
- No support for seamlessly mixing compiled and interpreted functions

➔ Interpreter not useful in same way for development

LFE shell

- Simple REPL
- Builtin variables `+ ++ +++ - * ** ***`
- Can set variables with `(set pat expr)`
- `(slurp file)` to load file and interpret all functions and macros
- Cannot define functions and macros (yet)
- No `(spit file)` yet either

LFE features

- The usual good lisp stuff – sexprs, macros, code ⇔ data
- Extensive use of pattern matching
- Uses Erlang data types and builtin functions
- Built on small core extended with macros
- Compiler, interpreter and shell

The BIG question

Apart from the Answer to Life, the Universe, and Everything

Will LFE end the complaints and moaning about Erlang syntax?

The Answer

42

NO!

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

LFE

Github: http://github.com/rvirding/lfe

Google groups:

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