The evolution of the Erlang VM

Joe Armstrong
Robert Virding
1985 - 1998
Pre history

AXE - programmed in PLEX

PLEX
  Programming language for exchanges)
  Proprietary
  blocks (processes) and signals
  in-service code upgrade

Eri Pascal
Phoning philosophers

Conclusion - Concurrent Logic programming with channel communication

Armstrong, Elshiewy, Virding (1986)
The Telephony Algebra - (1985)

idle(N) means the subscriber N is idle
on(N) means subscribed N in on hook

+\text{t}(A, \text{dial\_tone}) \text{ means add a dial tone to } A

\text{process}(A, f) :- \text{on}(A), \text{idle}(A), +\text{t}(A, \text{dial\_tone}),
+\text{d}(A, []), -\text{idle}(A), +\text{of}(A)

Using this notation, POTS could be described using fifteen rules. There was just one major problem: the notation only described how one telephone call should proceed. How could we do this for thousands of simultaneous calls?
The reduction machine - (1985)

A → B, C, D.
B → x, D.
D → y.
C → z.

A, B, C, D = nonterminals
x, y, z = terminals

To reduce X,...Y...
If X is a nonterminal replace it by its definition
If X is a terminal execute it and then do ...Y...
Aside - term rewriting is tail recursive

\[
A \rightarrow x, y, A \\
A \\
x, y, A \\
y, A \\
A \\
x, y, A \\
y, A \\
A \\
\ldots
\]

\[
\text{loop}(X) \rightarrow \\
\ldots \\
\text{loop}(X). \\
\]
factorial

rule(fac, 0) -> [pop, {push, 1}];
rule(fac, _) -> [dup, {push, 1}, minus, {call, fac}, times].

run() -> reduce0([[call, fac]], [3]).

reduce0(Code, Stack) ->
  io:format("Stack:~p Code:~p~n", [Stack, Code]),
  reduce(Code, Stack).

reduce([], [X]) -> X;
reduce([[push, N]|Code], T) -> reduce0(Code, [N|T]);
reduce([pop|Code], T) -> reduce0(Code, tl(T));
reduce([dup|Code], [H|T]) -> reduce0(Code, [H,H|T]);
reduce([minus|Code], [A,B|T]) -> reduce0(Code, [B-A|T]);
reduce([times|Code], [A,B|T]) -> reduce0(Code, [A*B|T]);
reduce([[call, Func]|Code], [H|_] = Stack) ->
  reduce0(rule(Func, H) ++ Code, Stack).
factorial

> fac:run().
Stack:[3] Code:[{call,fac}]
Stack:[3] Code:[{push,1},minus,{call,fac},times]
Stack:[3,3] Code:[{push,1},minus,{call,fac},times]
Stack:[1,3,3] Code:[{call,fac},times]
Stack:[2,3] Code:[{call,fac},times]
Stack:[2,3] Code:[{push,1},minus,{call,fac},times,times]
Stack:[2,2,3] Code:[{push,1},minus,{call,fac},times,times]
Stack:[1,2,2,3] Code:[{call,fac},times,times]
Stack:[1,2,3] Code:[{call,fac},times,times]
Stack:[1,2,3] Code:[{push,1},minus,{call,fac},times,times,times]
Stack:[1,1,2,3] Code:[{push,1},minus,{call,fac},times,times,times]
Stack:[1,1,1,2,3] Code:[{call,fac},times,times,times]
Stack:[0,1,2,3] Code:[{call,fac},times,times,times]
Stack:[0,1,2,3] Code:[{push,1},times,times,times,times]
Stack:[1,2,3] Code:[{push,1},times,times,times]
Stack:[1,1,2,3] Code:[{call,fac},times,times,times]
Stack:[1,2,3] Code:[{times,times,times}]
Stack:[2,3] Code:[{times}]
Stack:[6] Code:[]

787
Kreds/sec
1985 - 1989

Timeline

- Programming POTS/LOTS/DOTS (1885)
- A Smalltalk model of POTS
- A telephony algebra (math)
- A Prolog interpreter for the telephony algebra
- Added processes to prolog
- Prolog is too powerful (backtracking)
- Deterministic prolog with processes
- “Erlang” !!! (1986)
- ...
- Compiled to JAM code (1989)
- ...
erlang vsn 1.05

h help
reset reset all queues
reset_erlang kill all erlang definitions
load(F) load erlang file <F>.erlang
load load the same file as before
load(?) what is the current load file
what_erlang list all loaded erlang files
go reduce the main queue to zero
send(A,B,C) perform a send to the main queue
send(A,B) perform a send to the main queue
cq see queue - print main queue
wait_queue(N) print wait_queue(N)
cf see frozen - print all frozen states
eqns see all equations
eqn(N) see equation(N)
start(Mod,Goal) starts Goal in Mod
top top loop run system
q quit top loop
open_dots(Node) opens Node
talk(N) N=1 verbose, =0 silent
peep(M) set peeping point on M
no_peep(M) unset peeping point on M
eqns number is X

The manual 1986 (or 85)
Running a program

```erlang
joe> cat test.erlang
module(test).
1: start --> write('hello'),nl,go.
2: go --> start_proc(foo1,test,test),start_proc(foo2,test,test).
3: test --> wait.
4: wait,[X,1].
5: wait,[X,Y] --> write(received(Y)),nl,wait.

joe> erlang
erlang vsn 1.05
start erlang
type h for help

yes
| ?- load(test).
translating the file: test.erlang
Module: test
12345
compiling the file: test.obj
[/u/joe/logic/quintus/erlang/dots/test.obj compiled (1.950 sec 480 bytes)]
loading completed ...
```
The Prolog interpreter (1986)

version 1.06
dated
1986-12-18

1.03 “lost in the mists of time”
1988 - Interpreted Erlang

- 4 days for a complete re-write
- 245 reductions/sec
- semantics of language worked out
- Robert Virding joins the “team”
1989 - The need for speed

ACS- Dunder
- “we like the language but it’s too slow” - must be 40 times faster

Mike Williams writes the emulator (in C)

Joe Armstrong writes the compiler

Robert Virding writes the libraries
How does the JAM work?

• JAM has thee global data areas
  
  code space + atom table + scheduler queue

• Each process has a stack and a heap

• Erlang data structures are represented as tagged pointers on the stack and heap
JAM

- Compile code into sequences of instructions that manipulate data structures stored on the stack and heap (Joe)
- Write code loader, scheduler and garbage collector (Mike)
- Write libraries (Robert)
Atoms: example 'abc'

Integers: example 42

Tuples: \{abc,42,\{10,foo\}\}

Tagged Pointers
foo() -> {abc, 10}.

pushAtom abc

pushInt, 10

mkTuple, 2

Atom table

stack:

pushAtom abc

pushInt, 10

mkTuple, 2

heap:

stack:
Compiling foo() -> {abc,10}

switch(*pc++){
    case 16: // push short int
        *stop++ = mkint(*pc++);
        break;
    case 20: // mktuple
        arity = *pc++;
        *htop++ = mkarity(arity);
        while(arity>0){
            *htop++ = *stop--;
            arity--;
        }
        break;
}

{enter, foo,2}
{pushAtom, “abc”}
{pushInt, 10},
{mkTuple, 2},
ret

Byte code
16,10,20,2

pc = program counter
stop = stack top
htop = heap top

Part of the byte code interpreter
An early JAM compiler (1989)

Like the WAM with added primitives for spawning processes and message passing.
factorial

rule(fac, 0) -> [pop, {push, 1}];
rule(fac, _) -> [dup, {push, 1}, Minus, {call, fac}, times].

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

{info, fac, 1}
{try_me_else, label1}
  {arg, 0}
  {getInt, 0}
  {pushInt, 1}
  ret
label1: try_me_else_fail
  {arg, 0}
dup
  {pushInt, 1}
  minus
  {callLocal, fac, 1}
times
  ret
  ret
Jam improvements

- Uncessary stack -> heap movements
- Better with a register machine
- Convert to register machine by emulating top N stack locations with registers
- And a lot more ...
Alternate implementations

VEE (Virding's Erlang Engine)

- Experiment with different memory model
  - Single shared heap with real-time garbage collector (reference counting)
- Blindingly fast message passing

BUT

- No overall speed gain and more complex internals
Alternate implementations

Strand88 machine

• An experiment using another HLL as “assembler”
• Strand88 a concurrent logic language - every reduction a process and messages as cheap as lists
• Problem was to restrict parallelism

BUT

• Strand's concurrency model was not good fit for Erlang
• Worked but not as well as the JAM
Speedups

- Prolog Erlang Interpreter (1988) - 245 reds/sec
- Prolog JAM emulator - 35 reds/sec
- C Erlang JAM emulator (1989) - 30K reds/sec
- C Erlang BEAM emulator (2010) - 9 Mega reds/sec
- Erlang JAM emulator (2010) - 787K reds/sec
- Speedup $787K/35 = 22400$ in 21 years
- $K^{21} = 22400$ so $K = 1.61$ (61% / year) Smartness
- or $K^{21} = 767K/30K = 1.16$ (16% / year) Mores law
Links

A is linked to B
B is linked to C

If any process crashes an EXIT message is sent to the linked processes

This idea comes from the “C wire” in early telephones (ground the C wire to cancel the call)

Encourages “let it crash” programming
By 1990 things were going so well that we could ...

Buy a train set
We added new stuff

- Distribution
- OTP structure
- BEAM
- HIPE
- Type tools
- Philosophy

- Bit syntax
- Compiling pattern matching
- OTP tools
- Documented way of doing things
TEAM

Turbo Erlang Abstract Machine
By Bogumil Hausman

• Make a new efficient implementation of Erlang
TEAM

- New machine design
- Register machine
- Generate native code by smart use of GNU CC
- Same basic structures and memory design as JAM

```prolog
append([], X) -> X.
append([H|T], X) -> [H|append(T, X)];
append_2:
Clause;
TestNonEmptyList(x(0), next);
Allocate(1);
GetList2(x(0), y(0), x(0));
Call(append_2, 2);
TestHeap(2);
PutList2(x(0), y(0), x(0));
Deallocate(1);
Return;
ClauseEnd;
Clause;
TestNil(x(0), next);
Move(x(1), x(0));
Return;
ClauseEnd;
ErrorAction(FunctionClause);
```
TEAM

• Significantly faster than the JAM

BUT

• Module compilation was slow

• Code explosion, resultant code size was too big for customers

SO

• Hybrid machine with both native code and emulator
TEAM --> BEAM

Bogdan's Erlang Abstract Machine
And lots of improvements have been made and
lots of good stuff added!

Better GC (generational), SMP, NIFs, etc. etc.
- Pattern matching over bits

unpack(<<Red:5,Green:6,Blue:5>>) ->

...  

Due to Klacke  
(Claes Vikström)
Compiling pattern matching

• Erlang semantics say match clauses sequentially

BUT

• Don't have to if you are smart!

• Can group patterns and save testing

The Implementation of Functional Languages

Simon Peyton Jones

(old, from 1987, but still full of goodies)
Compiling pattern matching

scan1([\$\sl\Cs], St, Line, Col, Toks) when St#erl_scan.ws ->
scan1([\$\sl\Cs], St, Line, Col, Toks) ->
scan1([\$\nl\Cs], St, Line, Col, Toks) when St#erl_scan.ws ->
scan1([\$\nl\Cs], St, Line, Col, Toks) ->
scan1([\Cl\Cs], St, Line, Col, Toks) when C >= $A, C <= $Z ->
scan1([\Cl\Cs], St, Line, Col, Toks) when C >= $a, C <= $z ->

%% Optimization: some very common punctuation characters:
scan1([\$,\l\Cs], St, Line, Col, Toks) ->
scan1([\$(\l\Cs], St, Line, Col, Toks) ->
Compiling pattern matching

expr({var, Line, V}, Vt, St) →
expr({char, _Line, _C}, _Vt, St) → {[], St};
expr({integer, _Line, _I}, _Vt, St) → {[], St};
expr({float, _Line, _F}, _Vt, St) → {[], St};
expr({atom, _Line, A}, _Vt, St) →
expr({string, _Line, _S}, _Vt, St) → {[], St};
expr({nil, _Line}, _Vt, St) → {[], St};
expr({cons, _Line, H, T}, Vt, St) →
expr({lc, _Line, E, Qs}, Vt0, St0) →
expr({bc, _Line, E, Qs}, Vt0, St0) →
expr({tuple, _Line, Es}, Vt, St) →
expr({record_index, Line, Name, Field}, _Vt, St) →
expr({bin, _Line, Fs}, Vt, St) →
expr({block, _Line, Es}, Vt, St) →
expr({'if', Line, Cs}, Vt, St) →
expr({'case', Line, E, Cs}, Vt, St0) →
The Erlang VM as an assembler

• Efene
  • Mariano Guerra

• Reia
  • Tony Arcieri
    • http://wiki.reia-lang.org/wiki/Reia_Programming_Language

• LFE (Lisp Flavoured Erlang)
  • http://github.com/rvirding/lfe
The End