Pre-history

• AXE programmed in PLEX
• PLEX
  - programming language for exchanges
  - proprietary
  - blocks (processes) and signals
  - in-service code upgrade
• Eri-Pascal
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)
- ...
The telephony algebra (1985)

idle(N) means subscriber N is idle
on(N) means subscriber N is on hook
...
+t(N, dial_tone) means add dial tone to A

process(A, f) :- on(A), idle(A), +t(A,dial_tone),
                +d(A, []), -idle(A), +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

A -> B,C,D.
B -> x,D.
D -> y.
C -> z.

A
B, C, D
x, D, C, D
D, C, D
y, C, D
C, D
z, D
D
Y
{}

We can interrupt this at any time

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...
Term rewriting is last-call optimised

A -> x, y, B
B -> z, A

one(X0) ->
  ...
  two(X1).

two(Y0) ->
  ...
  one(Y1).
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
vsn(X)             erlang vsn number is X

The manual
1985 (or 86)
Running a program

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
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

Earlier versions
“lost in the mists of time”
Phoning philosophers

The Phoning Philosopher's Problem or Logic Programming for Telecommunications Applications

Armstrong, Elshiewy, Virding (1986)
1988 - Interpreted Erlang

- 4 days for a compiler rewrite
- 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? (1)

- JAM has three global data areas  
  code space + atom table + scheduler queue
- Each process has a stack and a heap  
  - fast context switching  
  - non-disruptive garbage collection
- Erlang data structures are represented as tagged pointers on the stack and heap
Atoms: example 'abc'

Integers: example 42

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

Tagged Pointers
How does the JAM work? (2)

- 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)
Factorial

\[
\begin{align*}
\text{rule}(\text{fac}, 0) & \rightarrow [\text{pop}, \{\text{push}, 1\}] ; & \text{fac}(0) & \rightarrow 1 ; \\
\text{rule}(\text{fac}, \_ ) & \rightarrow [\text{dup}, \{\text{push}, 1\}, \text{minus}, \{\text{call}, \text{fac}\}, \text{times}] . & \text{fac}(N) & \rightarrow N \times \text{fac}(N-1) .
\end{align*}
\]

\[
\text{run()} \rightarrow \text{reduce0}([\{\text{call}, \text{fac}\}], [3]).
\]

\[
\text{reduce0}(\text{Code}, \text{Stack}) \rightarrow \\
\text{io:format("Stack:\~p Code:\~p\n", [Stack, Code]),} \\
\text{reduce}(\text{Code}, \text{Stack}).
\]

\[
\begin{align*}
\text{reduce}([], [X]) & \rightarrow X ; \\
\text{reduce}([\{\text{push}, N\} | \text{Code}], T) & \rightarrow \text{reduce0}(\text{Code}, [N | T]) ; \\
\text{reduce}([\text{pop} | \text{Code}], [\_ | T]) & \rightarrow \text{reduce0}(\text{Code}, T) ; \\
\text{reduce}([\text{dup} | \text{Code}], [H | T]) & \rightarrow \text{reduce0}(\text{Code}, [H, H | T]) ; \\
\text{reduce}([\text{minus} | \text{Code}], [A, B | T]) & \rightarrow \text{reduce0}(\text{Code}, [B - A | T]) ; \\
\text{reduce}([\text{times} | \text{Code}], [A, B | T]) & \rightarrow \text{reduce0}(\text{Code}, [A \times B | T]) ; \\
\text{reduce}([\{\text{call}, \text{Func}\} | \text{Code}], [H | \_]=\text{Stack}) & \rightarrow \\
\text{reduce0}(\text{rule}(\text{Func}, H) \_ \_ \text{Code}, \text{Stack}).
\end{align*}
\]
Factorial

> fac:run().
Stack: [3] Code: [{call, fac}]
Stack: [3] Code: [dup, {push, 1}, minus, {call, fac}, times]
Stack: [3, 3] Code: [{push, 1}, minus, {call, fac}, times]
Stack: [1, 3, 3] Code: [minus, {call, fac}, times]
Stack: [2, 3] Code: [{call, fac}, times]
Stack: [2, 3] Code: [dup, {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: [minus, {call, fac}, times, times]
Stack: [1, 2, 3] Code: [{call, fac}, times, times]
Stack: [1, 2, 3] Code: [dup, {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: [minus, {call, fac}, times, times, times]
Stack: [0, 1, 2, 3] Code: [{call, fac}, times, times, times]
Stack: [0, 1, 2, 3] Code: [pop, {push, 1}, times, times, times]
Stack: [1, 2, 3] Code: [{push, 1}, times, times, times]
Stack: [1, 1, 2, 3] Code: [times, times, times]
Stack: [1, 2, 3] Code: [times, times]
Stack: [2, 3] Code: [times]
An early JAM compiler (1989)

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

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

{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
Compiling `foo() -> {abc,10}.` (1)

```erlang
\{enter, foo, 2\}
\{pushAtom, "abc"\}
\{pushInt, 10\},
\{mkTuple, 2\},
ret
```

Byte code

```
16,10,20,2
```

```
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;
}
```

pc = program counter
stop = stack top
htop = heap top
foo() -> {abc, 10}.

pushAtom abc

stack:
- a

pushInt, 10

stack:
- i
- 10
- a

mkTuple, 2

stack:
- T

heap:

<table>
<thead>
<tr>
<th>A</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>2</td>
</tr>
<tr>
<td>i</td>
<td>10</td>
</tr>
</tbody>
</table>

Atom table:

<table>
<thead>
<tr>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>a b c</td>
</tr>
</tbody>
</table>
An early JAM compiler (1989)

sys_sys.erl                 18 dummy
sys_parse.erl              783 erlang parser
sys_ari_parser.erl         147 parse arithmetic expressions
sys_build.erl              272 build function call arguments
sys_match.erl              253 match function head arguments
sys_compile.erl            708 compiler main program
sys_lists.erl               85 list handling
sys_dictionary.erl          82 dictionary handler
sys_utils.erl               71 utilities
sys_asm.erl                419 assembler
sys_tokenise.erl           413 tokeniser
sys_parser_tools.erl        96 parser utilities
sys_load.erl               326 loader
sys_opcodes.erl            128 opcode definitions
sys_pp.erl                 418 pretty printer
sys_scan.erl                252 scanner
sys_boot.erl               59 bootstrap
sys_kernel.erl             9 kernel calls
18 files                  4544

Like the WAM with added primitives for spawning processes and message passing
JAM improvements

• Unnecessary 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 (Virdings Erlang Engine)

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

- Blindingly fast message passing

BUT

- Small 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
1985-1998
By 1990 things were going so well that we could ...

Buy a train set
We added new stuff

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

- Bit syntax
- Compiling pattern matching
- OTP tools
- Documented way of doing things
Turbo Erlang Abstract Machine
Bogumil Hausman

• Make a new efficient implementation of Erlang

*Turbo Erlang: Approaching the Speed of C*
TEAM

- New machine design
- Register machine
- Generate native code by smart use of GCC
- Same basic structures and memory design as JAM
- Threaded emulator

```erlang
append([H|T], X) -> [H|append(T, X)];
append([], X) -> X.
```
Compiling foo() -> {abc,10}. (2)

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

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

Byte code
Threaded code

16,10,20,2
0x45620,10,0x45780,2

... pushInt: // push short int
*stop++ = mkint(*pc++);
goto *pc++;
...

... mkTuple: // mktuple
arity = *pc++;
*htop++ = mkarity(arity);
while(arity>0){
  *htop++ = *stop--;
  arity--;
};
goto *pc++;

static void *lables[] = {
  ...
  &&pushInt,
  ...
  &&mkTuple,
  ...
};
TEAM

• Significantly faster than the JAM

BUT

• Module compilation slow
• Code explosion, resultant code size 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, NIF’s etc. etc.
(now Björn’s Erlang abstract Machine)
Compiling pattern matching

• Erlang semantics say match clauses sequentially
  BUT
• Don’t have to if you’re 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

\[
\text{scan1([\$s|Cs], St, Line, Col, Toks) when St\#erl\_scan.ws ->}
\]
\[
\text{scan1([\$s|Cs], St, Line, Col, Toks) ->}
\]
\[
\text{scan1([\$n|Cs], St, Line, Col, Toks) when St\#erl\_scan.ws ->}
\]
\[
\text{scan1([\$n|Cs], St, Line, Col, Toks) ->}
\]
\[
\text{scan1([C|Cs], St, Line, Col, Toks) when C \geq \$A, C \leq \$Z ->}
\]
\[
\text{scan1([C|Cs], St, Line, Col, Toks) when C \geq \$a, C \leq \$z ->}
\]
\[
\text{%% Optimisation: some very common punctuation characters:}
\]
\[
\text{scan1([$,|Cs], St, Line, Col, Toks) ->}
\]
\[
\text{scan1([$(|Cs], St, Line, Col, Toks) ->}
\]
Compiling pattern matching

```erlang
expr({var,Line,V}, Vt, St) ->
expr({char,_Line,_C}, _Vt, St) ->
expr({integer,_Line,_I}, _Vt, St) ->
expr({float,_Line,_F}, _Vt, St) ->
expr({atom,Line,I}, _Vt, St) ->
expr({string,_Line,_S}, _Vt, St) ->
expr({nil,_Line}, _Vt, 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

- LFE (Lisp Flavoured Erlang)
  - Robert Virding
  - http://github.com/rvirding/lfe

- Reia
  - Tony Arcieri
  - http://reia-lang.org/
Robert Virding, Erlang Solutions Ltd.
robert.virding@erlang-solutions.com