Erlang Solutions Ltd.

A History of the Erlang VM

Robert Virding

© 1999-2011 Erlang Solutions Ltd.

Pre-history

- AXE programmed in PLEX
- PLEX
  - programming language for exchanges
  - proprietary
  - blocks (processes) and signals
  - in-service code upgrade
- Eri-Pascal

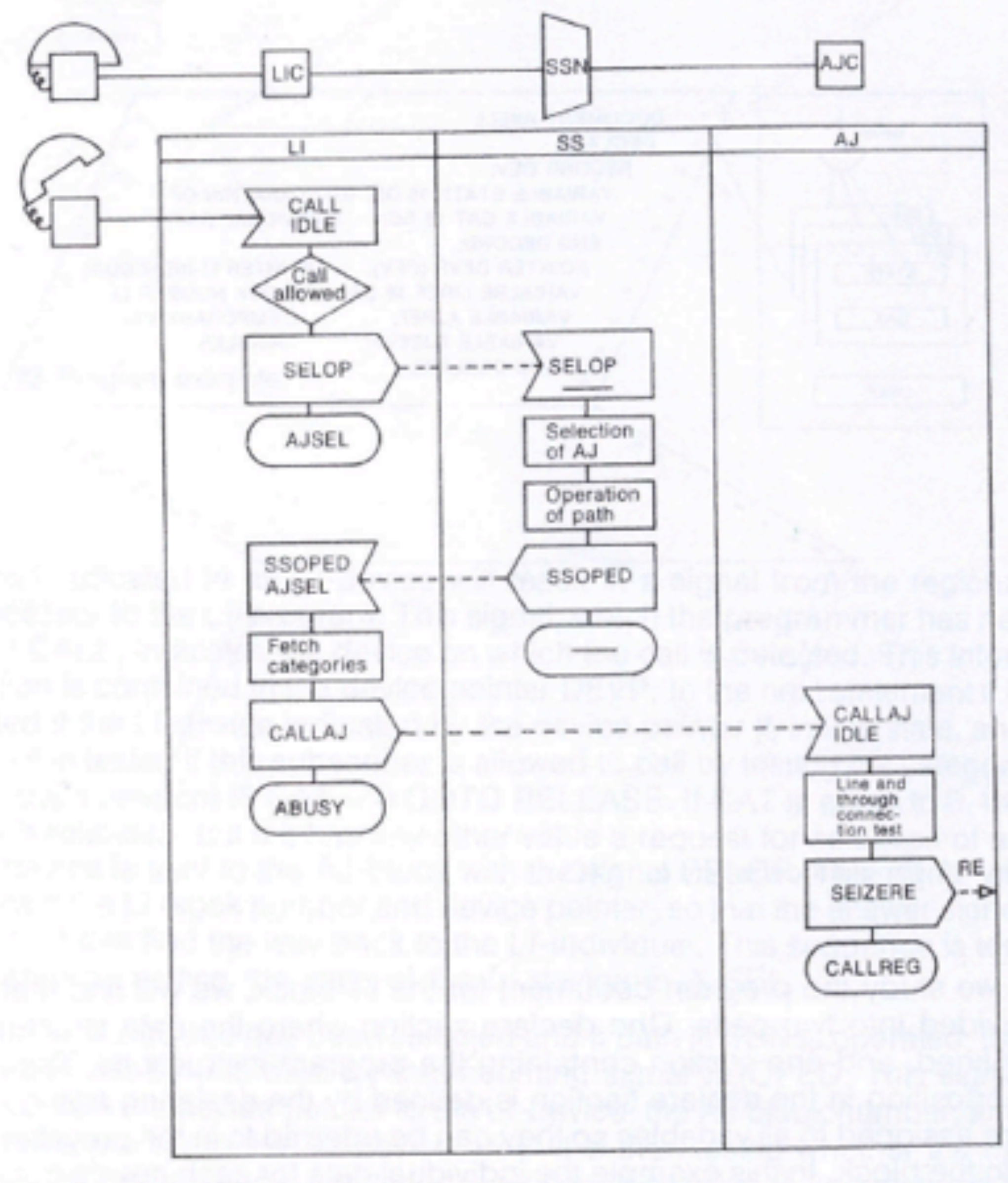


Fig. 11 AXE programming by PLEX

© 1999-2011 Erlang Solutions Ltd.

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 © 1999-2011 Erlang Solutions Ltd.

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 it's definition

If X is a terminal execute it and then do ...Y...

© 1999-2011 Erlang Solutions Ltd.

Term rewriting is last-call optimised

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

A
x,y,B
y,B
B
z,A
A
...
```

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

two(Y0) ->
    ...
    one(Y1).
```

© 1999-2011 Erlang Solutions Ltd.

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                    listing of program
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                              start erlang
erlang vsn 1.05
type h for help

yes
| ?- load(test).                         load  the program in test.erlang
translating the file:test.erlang
Module:test
12345                                    equantion numbers are displayed
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)

```
%    Package: make erlang
%    Author : Joseph Armstrong
%    Updated: 1986-12-18
%    Purpose: compiles and loads the erlang system


% this line MUST come first
:- ensure_loaded('/u/joe/logic/quintus/lib/set_library.pl').

%    vsn 1.03 lost in the mists of time
%    vsn 1.04 added modules and peeping (removed tracing)
%    vsn 1.05 mean version - fails in top loop to conserve space

%    vsn 1.06
%        added process constants
%                added commands
%                start_proc(Id,Module,Goal,Process_constants)
%                        is similar to start_proc/3 with added
%                        Process_constans
%                        Process_constants are a list of pairs of the form
%                                [(Key,Val),(Key1,Val1),...]
%                pconst(Key,Val)
%                        looks up the value of the process constant
%                        with key Key - Binds result to Value or makes
%                        error messages
%        added table driven number analyser
%                anal(Seq,Res)
%                        given a dialled sequence Seq binds Res
%                        to one of [invalid,get_more_digits,matched(Reason)]

vsn(1.06).


:- ensure_loaded(library(prims)).
:- ensure_loaded(library(findall)).

:- ensure_loaded('erlang1.04').
:- ensure_loaded(run).
:- ensure_loaded(queue).
:- ensure_loaded(reduce).
:- ensure_loaded(resume).
:- ensure_loaded(timeout).
```

Version 1.06 dated 1986-12-18

Earlier versions "lost in the mists of time"

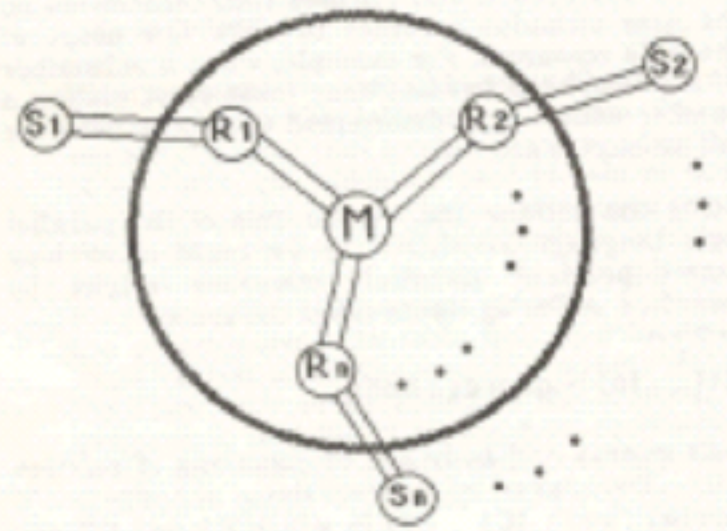
© 1999-2011 Erlang Solutions Ltd.

Phoning philosophers

(... guard : body ...).

7. A Telephone Exchange Model in PARLOG

Our exchange is modelled, in Parlog, as a set of communicating parallel logic processes, as illustrated in the figure below. Communication between logic processes takes place through unidirectional channels. A channel is represented by an infinite stream of messages.



The telephone sets are represented by external processes (Si's), each process (Si) communicates

State is an unbound variable which is bound to a value in the Manager process activation as follows:

```
manager([check_called(Rj,State)|To_M],
        From_M) :-
            get_state(Rj,State), ... .
```

in which the variable State gets a value to be bound in the caller_process communicating with the manager. This example is simplified a bit for illustration purposes. In the real program there are extra merging and forking processes to control communication to/from the manager.

An example of a time-dependent process is the hot-line service. The hot-line is a service provided by the exchange in which if a phone is picked up, and if no dialing has started within a given time, the system automatically dials a predefined number. This process is described in Parlog as follows:

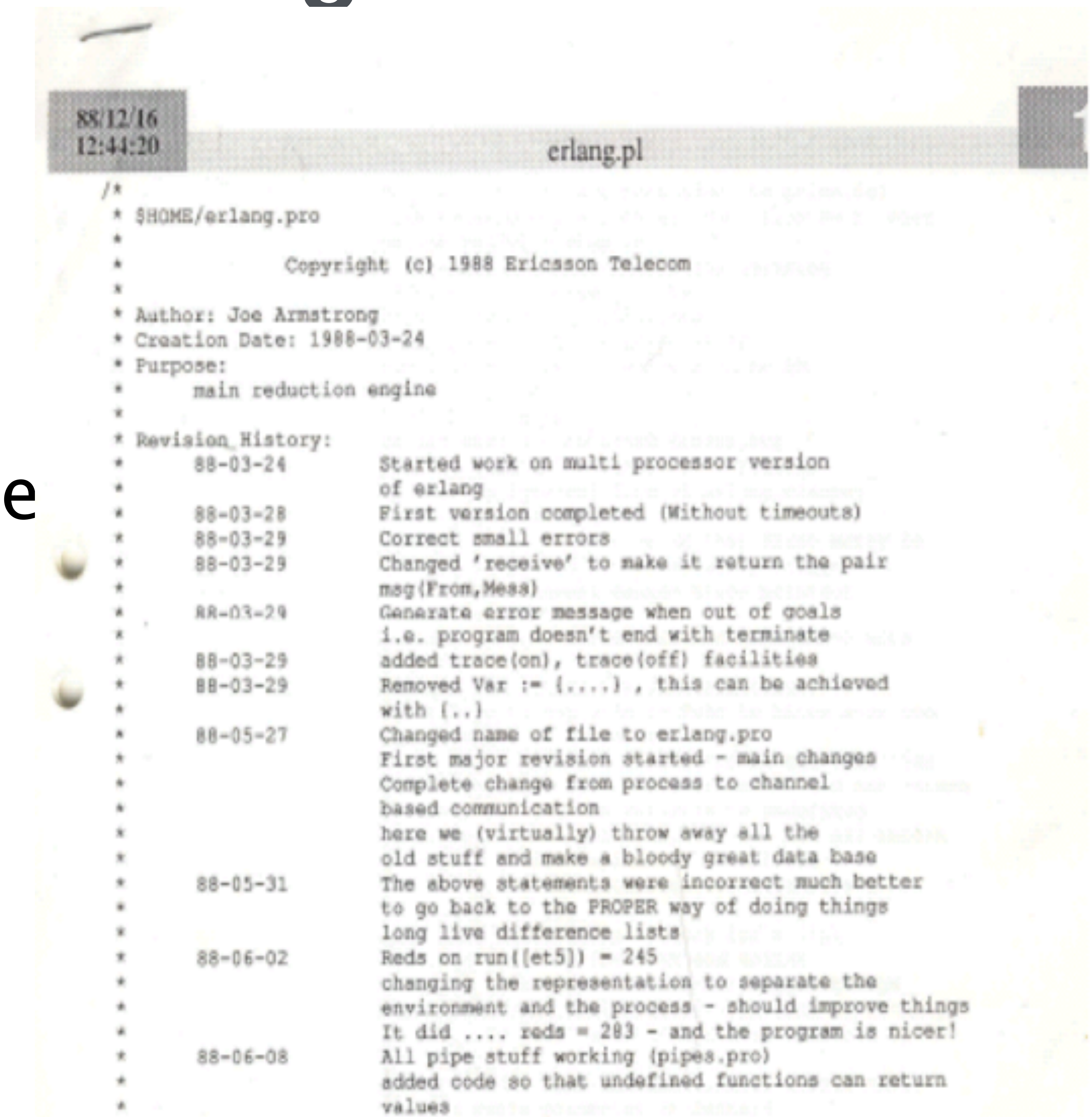
```
resource_process(Ri, [off_hook|From_S],
                 From_M, To_S, To_M) :-
    idle(Ri) :
        start_call(Ri, From_S, From_M, Alarm,
                   Stop_cmd, To_S, To_M),
        timer(some_time, Stop_cmd, Alarm).
```

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”



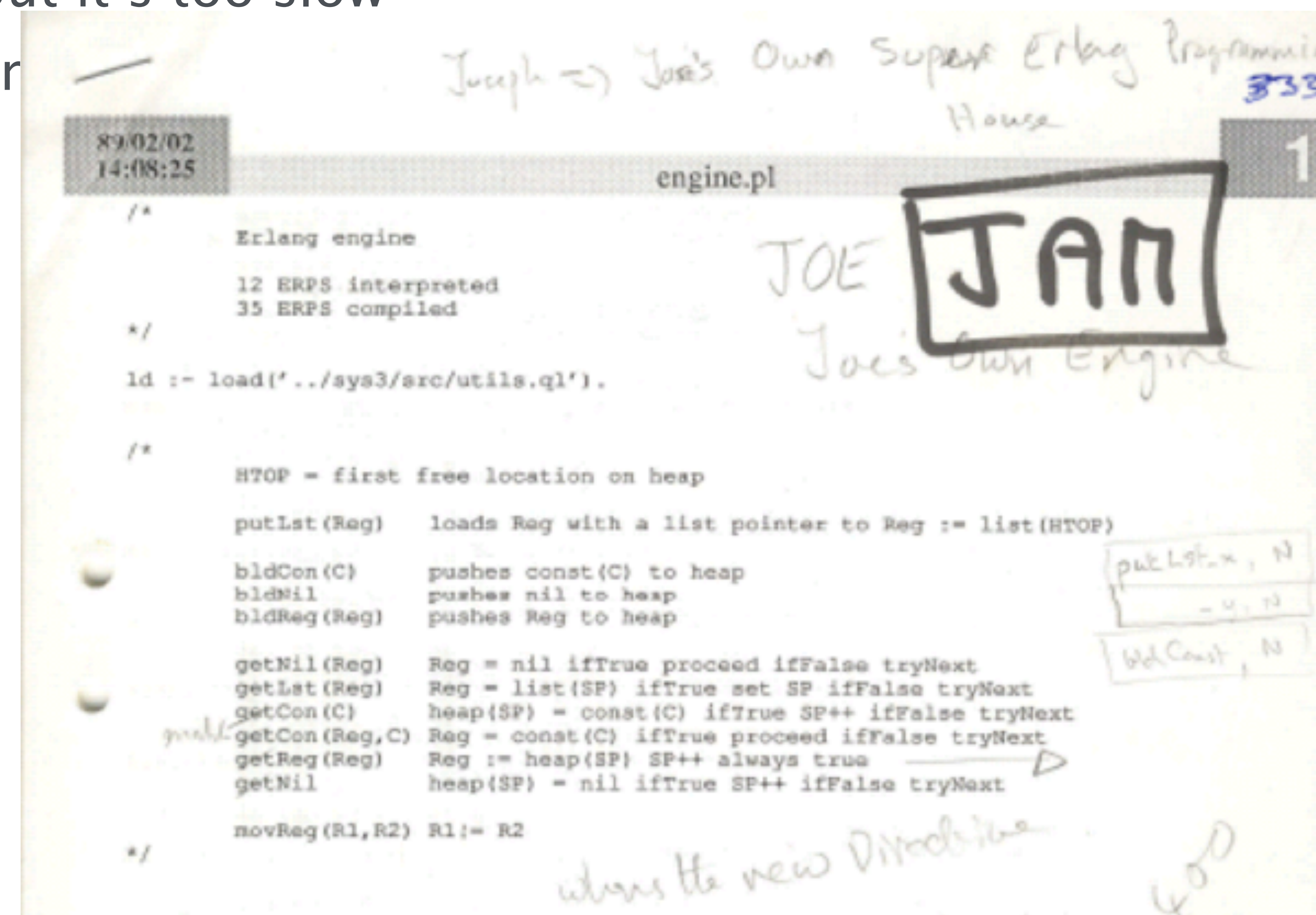
88/12/16
12:44:20 erlang.pl

```
/*
 * $HOME/erlang.pro
 *
 *              Copyright (c) 1988 Ericsson Telecom
 *
 * Author: Joe Armstrong
 * Creation Date: 1988-03-24
 * Purpose:
 *      main reduction engine
 *
 * Revision History:
 *      88-03-24        Started work on multi processor version
 *                      of erlang
 *      88-03-28        First version completed (Without timeouts)
 *      88-03-29        Correct small errors
 *      88-03-29        Changed 'receive' to make it return the pair
 *                      msg(From,Mess)
 *      88-03-29        Generate error message when out of goals
 *                      i.e. program doesn't end with terminate
 *      88-03-29        added trace(on), trace(off) facilities
 *      88-03-29        Removed Var := {....} , this can be achieved
 *                      with {..}
 *      88-05-27        Changed name of file to erlang.pro
 *                      First major revision started - main changes
 *                      Complete change from process to channel
 *                      based communication
 *                      here we (virtually) throw away all the
 *                      old stuff and make a bloody great data base
 *      88-05-31        The above statements were incorrect much better
 *                      to go back to the PROPER way of doing things
 *                      long live difference lists
 *      88-06-02        Reds on run([et5]) = 245
 *                      changing the representation to separate the
 *                      environment and the process - should improve things
 *                      It did .... reds = 283 - and the program is nicer!
 *      88-06-08        All pipe stuff working (pipes.pro)
 *                      added code so that undefined functions can return
 *                      values
```

© 1999-2011 Erlang Solutions Ltd.

1989 - The need for speed

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



89/02/02
14:08:25

engine.pl

1

```
/*
        Erlang engine

        12 ERPS interpreted
        35 ERPS compiled
*/

ld :- load('../sys3/src/utils.ql').

/*
        HTOP = first free location on heap

        putLst(Reg)     loads Reg with a list pointer to Reg := list(HTOP)

        bldCon(C)       pushes const(C) to heap
        bldNil          pushes nil to heap
        bldReg(Reg)     pushes Reg to heap

        getNil(Reg)     Reg = nil ifTrue proceed ifFalse tryNext
        getLst(Reg)     Reg = list(SP) ifTrue set SP ifFalse tryNext
        getCon(C)       heap(SP) = const(C) ifTrue SP++ ifFalse tryNext
        getCon(Reg,C)   Reg = const(C) ifTrue proceed ifFalse tryNext
        getReg(Reg)     Reg := heap(SP) SP++ always true
        getNil          heap(SP) = nil ifTrue SP++ ifFalse tryNext

        movReg(R1,R2)   R1:= R2
*/
```


- Mike Williams writes the emulator (in C)
- Joe Armstrong writes the compiler
- Robert Virding writes the libraries

© 1999-2011 Erlang Solutions Ltd.

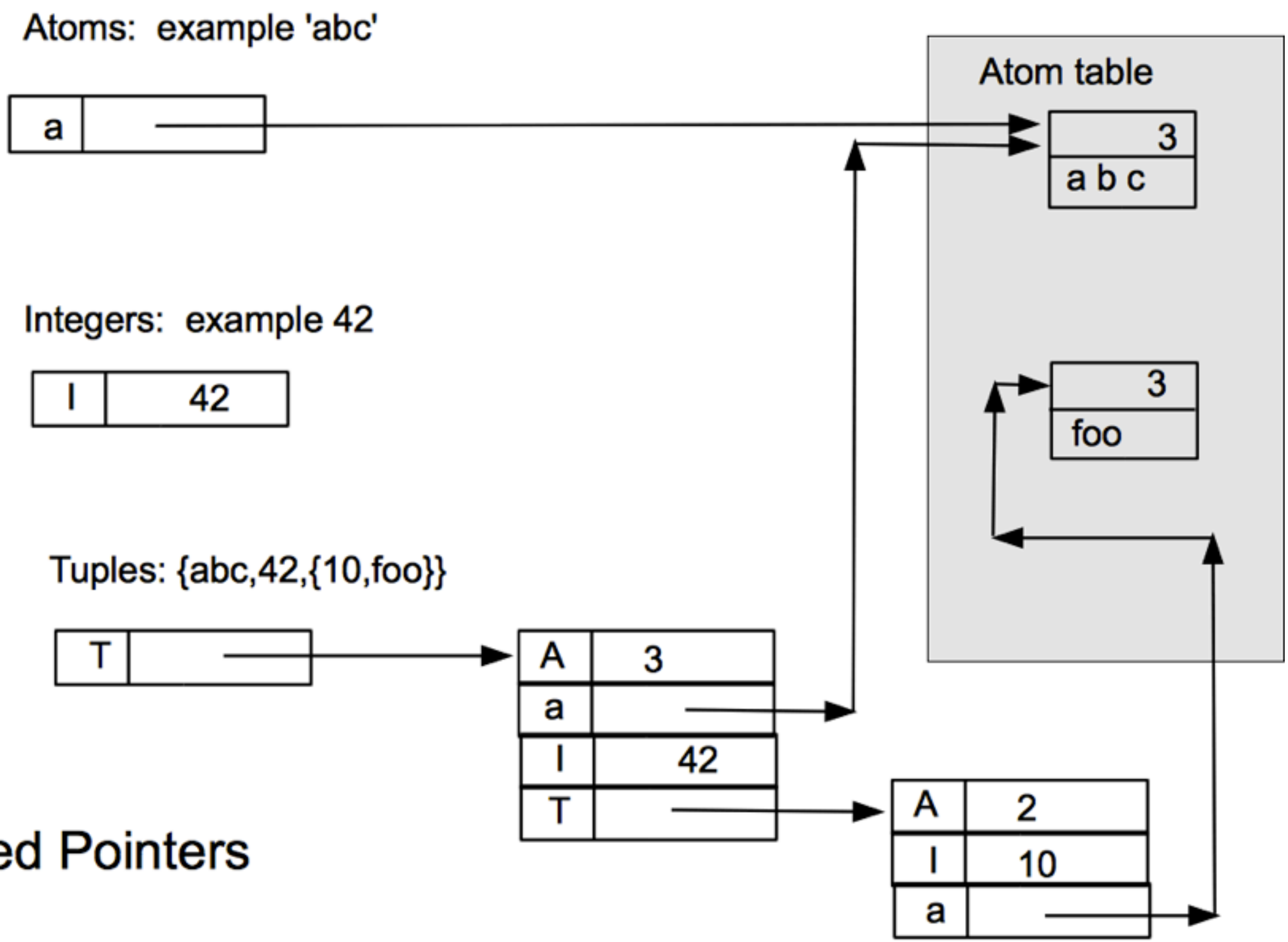
How does the JAM work? (1)

- 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

© 1999-2011 Erlang Solutions Ltd.

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)

© 1999-2011 Erlang Solutions Ltd.

Factorial

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

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, 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).
```

© 1999-2011 Erlang Solutions Ltd.

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]
Stack:[6] Code:[]
```

© 1999-2011 Erlang Solutions Ltd.

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
```


© 1999-2011 Erlang Solutions Ltd.

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

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

→

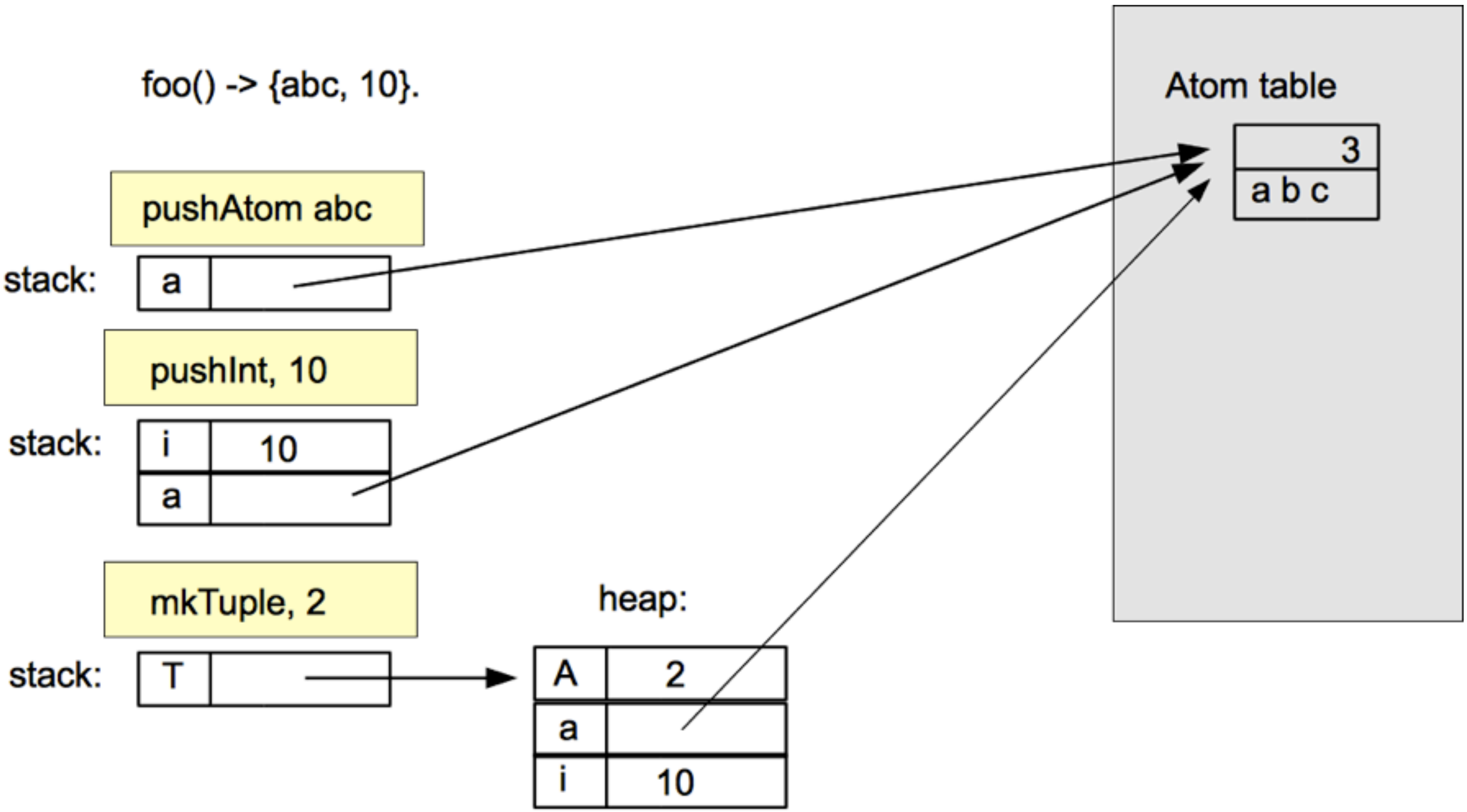
Byte code

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

© 1999-2011 Erlang Solutions Ltd.



© 1999-2011 Erlang Solutions Ltd.

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

Erlang

© 1999-2011 Erlang Solutions Ltd.

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 ...

© 1999-2011 Erlang Solutions Ltd.

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

© 1999-2011 Erlang Solutions Ltd.

Alternate implementations

Strand88 machine

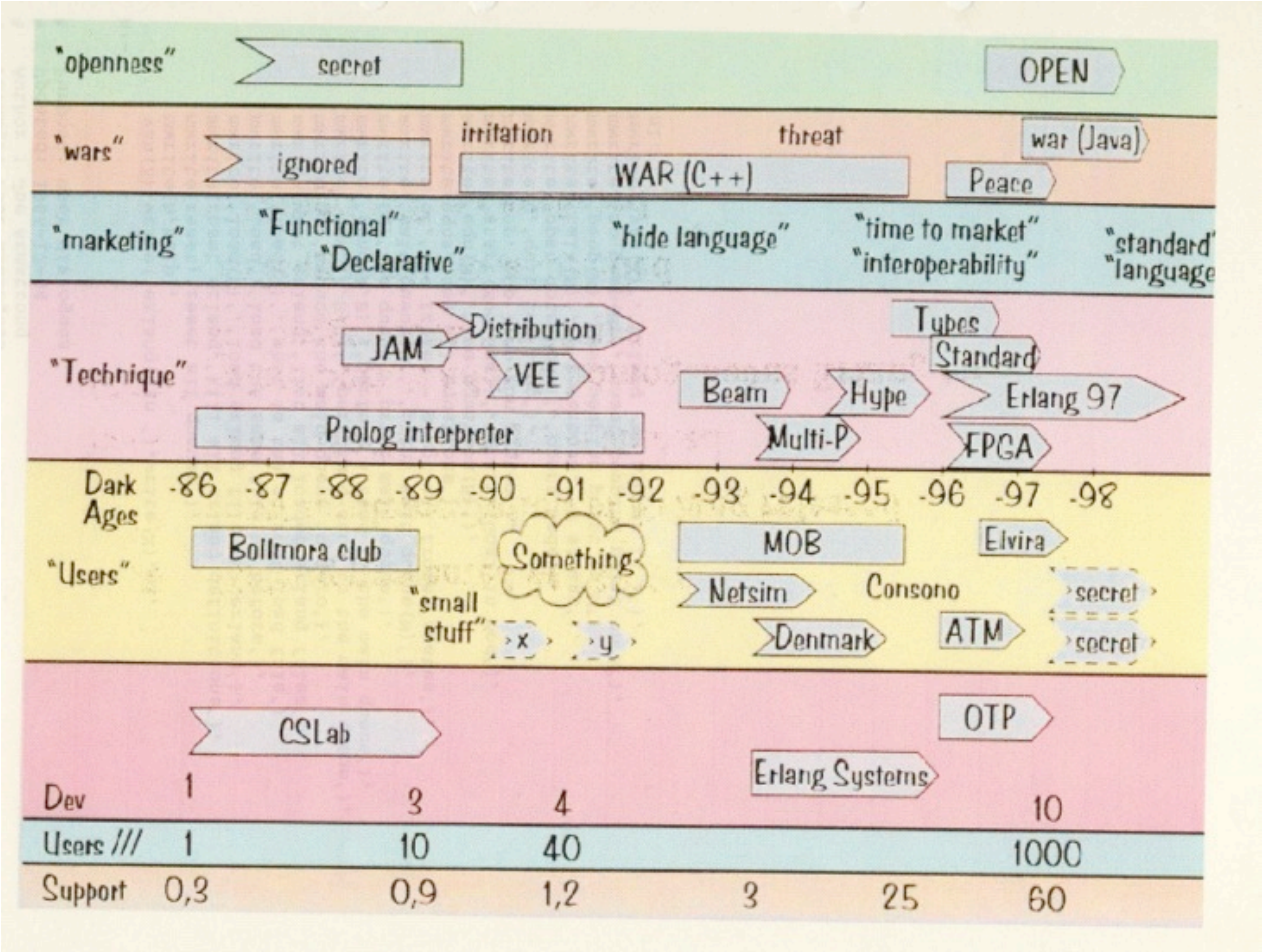
- An experiment using another HLL as “assembler”
- Strand88 a concurrent logic language
  - every reduction a process and messages as cheap as lists
- Problem was to restrict parallelism

BUT

- Strand's concurrency model was not good fit for Erlang

© 1999-2011 Erlang Solutions Ltd.

1985-1998

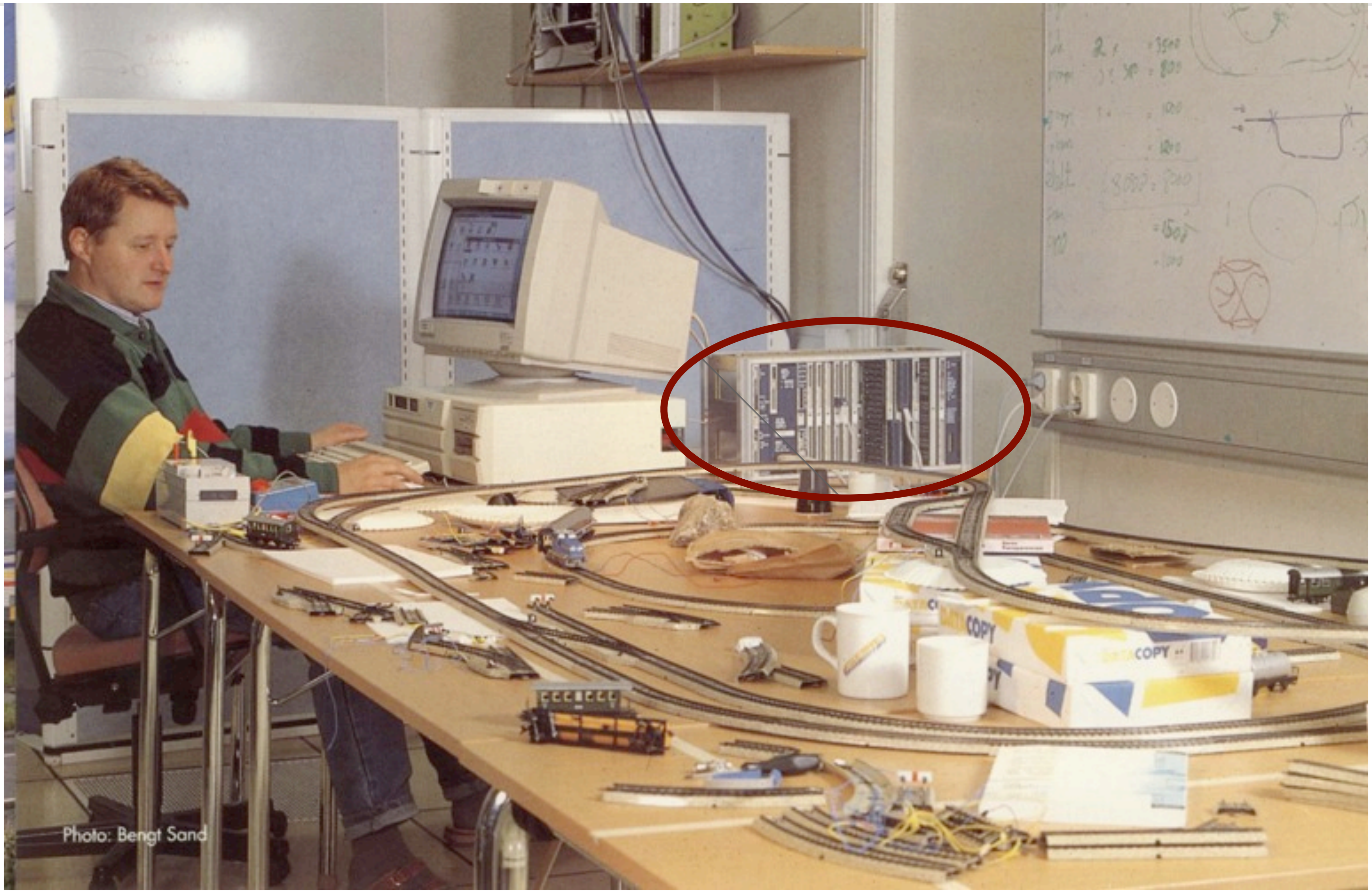


"openness"
secret
OPEN
"wars"
ignored
irritation
threat
WAR (C++)
war (Java)
Peace
"marketing"
"Functional"
"Declarative"
"hide language"
"time to market"
"interoperability"
"standard"
"language
"Technique"
Distribution
JAM
VEE
Prolog interpreter
Beam
Multi-P
Hype
Types
Standard
Erlang 97
FPGA
Dark Ages
-86 -87 -88 -89 -90 -91 -92 -93 -94 -95 -96 -97 -98
"Users"
Bollmora club
Something
"small stuff"
x
y
MOB
Netsim
Consono
Denmark
ATM
Elvira
secret
secret
CSLab
OTP
Erlang Systems
Dev 1 3 4 10
Users /// 1 10 40 1000
Support 0,3 0,9 1,2 3 25 60

By 1990 things
were going
so well
that we
could
...

© 1999-2011 Erlang Solutions Ltd.

Buy a train set



© 1999-2011 Erlang Solutions Ltd.

We added new stuff

- Distribution
- Philosophy
- OTP structure
- BEAM
- HIPE
- Type Tools
- Bit syntax
- Compiling pattern matching
- OTP tools
- Documented way of doing things

© 1999-2011 Erlang Solutions Ltd.

TEAM

Turbo Erlang Abstract Machine

Bogumil Hausman

- Make a new efficient implementation of Erlang

Turbo Erlang: Approaching the Speed of C

© 1999-2011 Erlang Solutions Ltd.

TEAM

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

```
append([H|T], X) -> [H|append(T, X);
append([], X) -> 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);
```


© 1999-2011 Erlang Solutions Ltd.

Compiling foo() -> {abc,10}. (2)

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

→ Byte code → Threaded code

16,10,20,2 → 0x45620,10,0x45780,2

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

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

```
...
pushInt:   // push short int
    *stop++ = mkint(*pc++);
    goto *pc++;
...
mkTuple:   // mktuple
    arity = *pc++;
    *htop++ = mkarity(arity);
    while(arity>0){
        *htop++ = *stop--;
        arity--;
    };
    goto *pc++;
```

© 1999-2011 Erlang Solutions Ltd.

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

© 1999-2011 Erlang Solutions Ltd.

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)

© 1999-2011 Erlang Solutions Ltd.

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)

© 1999-2011 Erlang Solutions Ltd.

Compiling pattern matching

```
scan1([$\s|Cs], St, Line, Col, Toks) when St#erl_scan.ws ->
scan1([$\s|Cs], St, Line, Col, Toks) ->
scan1([$\n|Cs], St, Line, Col, Toks) when St#erl_scan.ws ->
scan1([$\n|Cs], St, Line, Col, Toks) ->
scan1([C|Cs], St, Line, Col, Toks) when C >= $A, C =< $Z ->
scan1([C|Cs], St, Line, Col, Toks) when C >= $a, C =< $z ->
%% Optimisation: some very common punctuation characters:
scan1([$,|Cs], St, Line, Col, Toks) ->
scan1([$(|Cs], St, Line, Col, Toks) ->
```

© 1999-2011 Erlang Solutions Ltd.

Compiling pattern matching

```
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) ->
```

© 1999-2011 Erlang Solutions Ltd.

The Erlang VM as an assembler

- Efene
  - Mariano Guerra
  - http://marianoguerra.com.ar/efene/
- LFE (Lisp Flavoured Erlang)
  - Robert Virding
  - http://github.com/rvirding/lfe
- Reia
  - Tony Arcieri
  - http://reia-lang.org/

© 1999-2011 Erlang Solutions Ltd.

THE END

Robert Virding, Erlang Solutions Ltd.

robert.virding@erlang-solutions.com

© 1999-2011 Erlang Solutions Ltd.