JErlang: Erlang with Joins

Hubert Plociniczak

hp105@doc.ic.ac.uk

Supervisor: Professor Susan Eisenbach
Second marker: Professor Sophia Drossopoulou

June 22, 2009

Erlang

- Functional language
- Dynamic typing
- Concurrency Oriented Programming (Actors)
- Fault-tolerant, non-stop running
- Single variable assignment
- Powerful pattern-matching
- Open Telecom Platform (for example **gen_server** behaviour used for implementing the server in the client-server model)

Erlang - Processes and Communication

Create process

```
Pid = spawn(fun() -> io:format("New Process!", []))
```

Send message

```
Pid ! {john, salary, 22000}
```

Receive message

```
1 receive
2     {ok, Value} ->
3         process_value(Value);
4     {error, Reason} ->
5         shutdown(Reason)
6 after
7     Timeout -> process_no_response()
8 end
```

How to synchronise on more than one message? (Wrong)


```
1  receive
2      {get, X} ->
3          receive
4              {set, Y} when (X == Y) ->
5                  {found, 2, X}
6          end;
7      {set, X} ->
8          receive
9              {get, Y} when (X == Y) ->
10                 {found, 2, X}
11         end
12 end
```

How to synchronise on more than one message? (Almost)

```
1  fun() ->
2      A = fun(ReceiveFunc) ->
3              receive
4                  {get, X} ->
5                      receive
6                          {set, Y} when (X == Y) ->
7                              {found, 2, X}
8                      after 0 ->
9                          self() ! {get, X},
10                         ReceiveFunc(ReceiveFunc)
11                     end
12                 {set, X} ->
13                     receive
14                         {get, Y} when (X == Y) ->
15                             {found, 2, X}
16                     after 0 ->
17                         self() ! {set, X},
18                         ReceiveFunc(ReceiveFunc)
19                     end
20             end
21         end,
22         A(A)
23 end.
```

Join - calculus

- Based on a Abstract Chemical Machine used for chemical reactions
- Process Calculi developed at INRIA
- Equivalent to Π-calculus
- Introduces multi-way join patterns
- Non-deterministic choice

Syntax

function	::=	$\overline{functiondef}$
functiondef	|	$(\overline{p})$ **when** *guard* $\rightarrow$ *e*
expr(*e*)	::=	$e_1\ op_e\ e_2$
	|	$e(\overline{e})$
	|	**case** *e* **of** $\overline{match}$ **end**
	|	**receive** $\overline{join}$ **end**
	|	$e_1\ !\ e_2$
	|	$e_1\ ,\ e$
	|	$p =_{\mathbf{m}} e_2$
	|	*basicvalue* | *varId* | { $\overline{e}$ } | [$\overline{e}$]
basicvalue	::=	atom | number | pid | funcId
value(*v*)	::=	*basicvalue* | { $\overline{v}$ } | [$\overline{v}$]
pattern(*p*)	::=	*varId* | *basicvalue* | { $\overline{p}$ } | [$\overline{p}$]
match	::=	*p* **when** *g* $\rightarrow$ *e*
join	::=	*jpattern* [*join_aux*] [**when** *g*] $\rightarrow$ e
join_aux	::=	**and** *jpattern* [*join_aux*]
jpattern	::=	*propagation p*
propagation	::=	**true** | **false**
guard(*g*)	::=	$g_1\ op_g\ g_2$ | *basicvalue*
	|	*varId* | $g(\overline{g})$ | { $\overline{g}$ } | [$\overline{g}$]

Structural Operational Semantics

$$\text{Var}_0: \frac{\begin{array}{l} F(\mathbf{varId}) = v \\ v \neq \text{Udf} \end{array}}{\mathbf{varId},\ F,\ Q \rightsquigarrow_E\ v,\ F,\ Q}$$

$$\text{Seq}: \frac{}{(v,\ e),\ F,\ Q \rightsquigarrow_E\ e,\ F,\ Q}$$

$$\text{Receive}_1: \frac{\begin{array}{l} Q^a \leq_q Q \\ \forall (i \in 1..n)\ (join_i =_s jpattern_{i,1}\ \mathbf{and}\ \ldots\ jpattern_{i,n}\ \mathbf{when}\ g_i \rightarrow e_i) \\ k \in 1..n,\ joinMatches\ (join_k,\ F,\ Q,\ Q^a,\ F',\ Q') \end{array}}{\mathbf{receive}\ join_1\ \ldots\ join_n\ \mathbf{end},\ F,\ Q \rightsquigarrow_E\ e_k,\ F',\ Q'}$$

$$\text{Receive}_2: \frac{\forall (Q'' \leq_q Q).\forall (1 \leq l \leq n).\neg \exists F', Q' (joinMatches(join_l,\ F,\ Q'',\ F',\ Q'))}{\mathbf{receive}\ join_1\ \ldots\ join_n\ \mathbf{end},\ F,\ Q \rightsquigarrow_E\ \mathbf{error},\ F,\ Q}$$

Joins First Match Semantics

- There is a join which can be satisfied with the subset of the current mailbox
- For any previous join, with the subset of the mailbox given above, matching is unsuccessful
- For any smaller subset of the mailbox none of the defined joins can be satisfied

$$\text{Receive}_{First-Match}: \frac{\begin{array}{l} Q^a \leq_q Q \\ \forall(i \in 1..n)\ (join_i =_s jpattern_{i,1}\ \textbf{and}\ \ldots\ jpattern_{i,n}\ \textbf{when}\ g_i \rightarrow e_i) \\ k \in 1..n,\ joinMatches\ (join_k,\ F,\ Q,\ Q^a,\ F',\ Q') \\ \forall(Q^b <_q Q^a).\forall(1 \leq l \leq n).\neg\exists F'', Q'' \\ \qquad (joinMatches(join_l,\ F,\ Q,\ Q^b,\ F'',\ Q'')) \\ \forall(1 \leq l < k).\neg\exists F'', Q' \\ \qquad (joinMatches(join_l,\ F,\ Q,\ Q^a, F'',\ Q'')) \end{array}}{\textbf{receive}\ join_1\ \ldots\ join_n\ \textbf{end},\ F,\ Q \rightsquigarrow_E e_k,\ F',\ Q'}$$

JErlang language features

- Joins
- Guards
- Timeouts
- Non-linear patterns
- Propagation
- Synchronous calls
- new OTP design pattern

Joins

- First Match semantics
- The possibility of having multiple patterns (possibly the same)
- Take into account the context (bounded vs unbounded variables)
- Messages' order preserved

```
1  operation() ->
2      receive
3          {ok, sum} and {val, X} and {val, Y} ->
4              {sum, X + Y};
5          {ok, mult} and {val, X} and {val, Y} ->
6              {mult, X * Y};
7          {ok, sub} and {val, X} and {val, Y} ->
8              {sub, X - Y}
9      end
10 end.
```

Guards

Provides additional filtering not expressible in terms of patterns. Limited number of expressions without side-effects (Erlang constraint).

```
1 receive
2   {Transaction, M} and {limit, Lower, Upper}
3         when (Lower <= M and M <= Upper) ->
4     commit_transaction(M, Transaction)
5 end
```

Timeouts

- Not specified in Join-calculus
- Integral part of Erlang **receive** construct
- Time measured for *idle* periods

```
1  cash_machine_authorise(Timeout) ->
2      receive
3          {pin, Pin} and {card, Number} ->
4              authenticate(Pin, Number);
5          abort and {card, Number} ->
6              abort_authentication(Number)
7      after Timeout ->
8          take_card(Number, Timeout);
9      end
10 end.
```

Non-linear patterns

- Messages can match multiple joins
- Unbounded variables spanned over multiple patterns have to agree on values
- Computationally expensive

```
1  receive
2    {get, X} and {set, X} ->
3      {found, 2, X}
4  end
5  ...
6  receive
7    {Pin, Id} and {auth, Pin} and {commit, Id} ->
8      perform_transaction(Pin, Id)
9  end.
```

Propagation

Inspired by Constraint Handling Rules and Haskell Join Rules language.
Allows for copying correct messages instead of removing them.

```
1 receive
2   prop({session, Id}) and {act, Action, Id} ->
3       perform_action(Action, Id);
4   {session, Id} and {logout, Id} ->
5       logout_user(Id)
6 end
```

Synchronous calls

Explicit synchronisation requires process identifiers in the message.

```
1  receive
2    {accept, Pid1} and {asynchronous, Value}
3                   and {accept, Pid2}   ->
4        Pid1 ! {ok, Value},
5        Pid2 ! {ok, Value}
6  end
```

gen_joins

Open Telecom Platform design pattern (in comparison to **gen_server** allows for synchronisation on multiple messages)

```
1  get() ->
2    jerlang_gen_joins:call(?MODULE, get).
3
4  set(Value) ->
5    jerlang_gen_joins:cast(?MODULE, {set,Value}).
6
7  %% CALLBACKS
8  handle_join( {set, Value} and get, Status) ->
9    {[noreply, {reply, {ok, Value}}],
10       [Value | Status]}.
```

Definition (Santa Claus problem)

Santa Claus sleeps at the North pole until awakened by either all of the nine reindeer, or by a group of three out of ten elves. He performs one of two indivisible actions:

- If awakened by the group of reindeer, Santa harnesses them to a sleigh, delivers toys, and finally unharnesses the reindeer who then go on holiday.
- If awakened by a group of elves, Santa shows them into his office, consults with them on toy R&D, and finally shows them out so they can return to work constructing toys.

A waiting group of reindeer must be served by Santa before a waiting group of elves.

Santa Claus Demo

Santa Claus solution in **gen_joins**

Santa Claus - Minimal Erlang solution

```
1  worker(Secretary, Message) ->
2      receive after random:uniform(1000) -> ok end,  % random delay
3      Secretary ! self(),          % send my PID to the secretary
4      Gate_Keeper = receive X -> X end,   % await permission to enter
5      io:put_chars(Message),       % do my action
6      Gate_Keeper ! {leave,self()},   % tell the gate-keeper I'm done
7      worker(Secretary, Message).     % do it all again
8
9  secretary(Santa, Species, Count) ->
10     secretary_loop(Count, [], {Santa,Species,Count}).
11
12 secretary_loop(0, Group, {Santa,Species,Count}) ->
13     Santa ! {Species,Group},
14     secretary(Santa, Species, Count);
15 secretary_loop(N, Group, State) ->
16     receive PID ->
17         secretary_loop(N-1, [PID|Group], State)
18     end.
19
20 santa() ->
21     {Species,Group} =
22     receive              % first pick up a reindeer group
23         {reindeer,G} -> {reindeer,G}% if there is one, otherwise
24     after 0 ->
25             receive         % wait for reindeer or elves,
26                 {reindeer,G} -> {reindeer,G}
27               ; {elves,G}    -> {elves,G}
28             end          % whichever turns up first.
29     end,
30     case Species
31       of reindeer -> io:format("Ho, ho, ho! Let's deliver toys!~n", [])
32        ; elve -> io:format("Ho, ho, ho! Let's meet in the study!~n", [])
33     end,
34     [PID ! self() || PID <- Group], % tell them all to enter
35     [receive {leave,PID} -> ok end  % wait for each of them to leave
36     || PID <- Group],
37     santa().
38
39 spawn_worker(Secretary, Before, I, After) ->
40     Message = Before ++ integer_to_list(I) ++ After,
41     spawn(fun () -> worker(Secretary, Message) end).
```

Santa Claus - Minimal JErlang solution

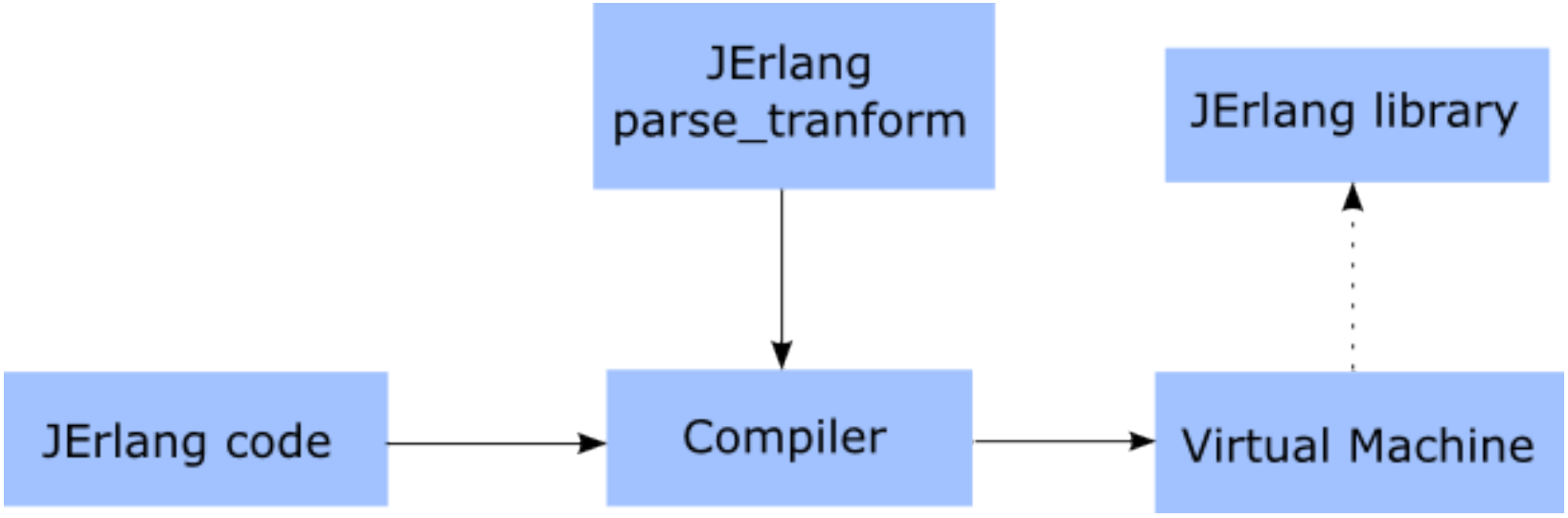
```
1  santa() ->
2      io:format("It was a long night. Time to bed~n"),
3      Group =
4      receive
5          {reindeer, Pid1} and {reindeer, Pid2} and
6          {reindeer, Pid3} and {reindeer, Pid4} and
7          {reindeer, Pid5} and {reindeer, Pid6} and
8          {reindeer, Pid7} and {reindeer, Pid8} and
9          {reindeer, Pid9} ->
10             io:format("Ho, ho, ho! Let's deliver presents!~n"),
11             [Pid1, Pid2, Pid3, Pid4,
12              Pid5, Pid6, Pid7, Pid8, Pid9];
13         {elf, Pid1} and {elf, Pid2} and {elf, Pid3} ->
14             io:format("Ho, ho, ho! Let's discuss R&D possibilites!~n"),
15             [Pid1, Pid2, Pid3]
16     end,
17     [ Pid ! ok || Pid <- Group],
18
19     %% Synchronise on return of the animals
20     receive
21         {r, done} and {r, done} and {r, done} and {r, done} and {r, done}
22         and {r, done} and r, done} and {r, done} and {r, done} ->
23             ok;
24         {e, done} and {e, done} and {e, done} ->
25             ok
26     end,
27     santa().
28
29 worker(Santa, Type, Id, Action, Ans) ->
30     worker1(Santa, Type, Id, Action, Ans).
31
32 worker1(Santa, Type, Id, Action, Ans) ->
33     receive after random:uniform(1000) -> ok end,
34     Santa ! {Type, self()},
35     io:format("~p ~p: Waiting at the gate~n", [Type, Id]),
36     receive ok -> ok end,
37     io:format("~p ~p: ~p~n", [Type, Id, Action]),
38     Santa ! {Ans, done},
39     worker1(Santa, Type, Id, Action, Ans).
```

Sad Santa Claus - Minimal JErlang solution

No additional “hacks” for proper prioritisation and clear solution for different types of messages

```
1  receive
2      {reindeer, Pid1} and {reindeer, Pid2} and {reindeer, Pid3}
3        and {reindeer, Pid4} and {reindeer, Pid5} and {reindeer, Pid6}
4        and {reindeer, Pid7} and {reindeer, Pid8} and {reindeer, Pid9} ->
5          io:format("Ho,ho,ho! Let's deliver presents!~n"),
6          [Pid1, Pid2, Pid3, Pid4,
7           Pid5, Pid6, Pid7, Pid8, Pid9];
8      {ork,sergant,Pid1} and {ork,captain,Pid2} and {ork,sergant,Pid3} ->
9          io:format("Ho,ho,ho? No presents, orks destroyed the factory!~n"),
10         [Pid1, Pid2, Pid3];
11     {elf, Pid1} and {elf, Pid2} and {elf, Pid3} ->
12         io:format("Ho,ho, ho!Let's discuss R&D possibilites!~n"),
13         [Pid1, Pid2, Pid3]
14 end
```

General architecture



parse_transform and Abstract Syntax Trees

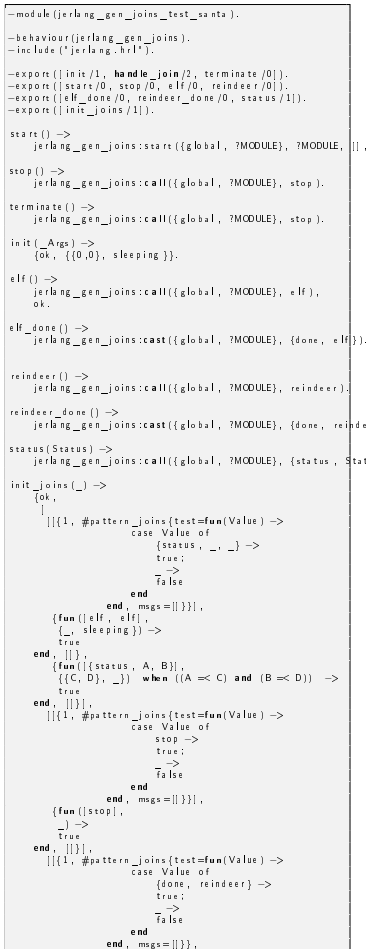
- Different semantics to the valid Erlang syntax code
- Requires only a single line of code in the original module

```
-compile({parse_transform, jerlang_parse}).
```

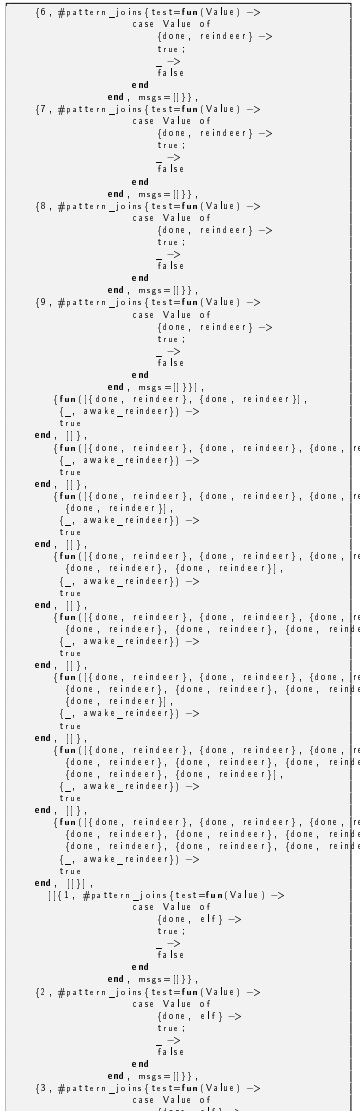
- Runs as part of the parser stage in the compiler
- JErlang provides various version of **parse_transform** for different modes
- Independent of the Erlang release, compiler, run-time and the application
- Uses familiar Erlang syntax for analysis of the code (pattern matching, tail recursion)
- Enables pretty joins syntax instead of complex calls to JErlang's library
- Required careful analysis of the code to avoid spurious warnings and errors
- "Not recommended" for standard Erlang programmers

Santa Claus (**gen_joins**) without parse_transform

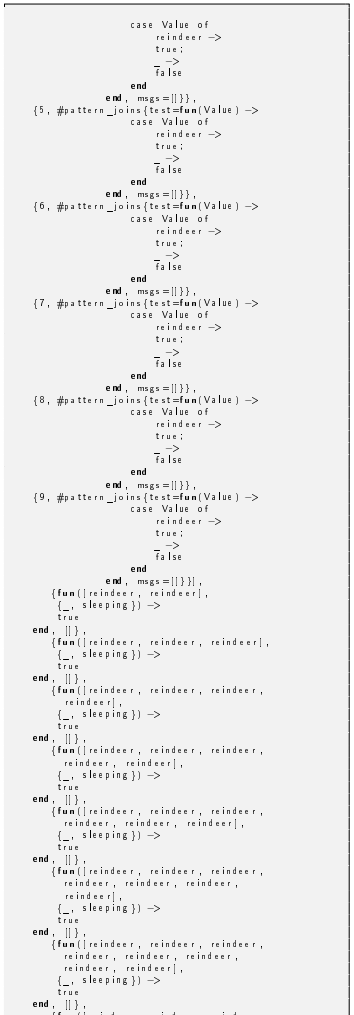
Almost 400 lines of code of condensed code



```
-module(jerlang_gen_joins_test_santa).

-behaviour(jerlang_gen_joins).
-include("jerlang.hrl").

-export([init/1, handle_join/2, terminate/0]).
-export([start/0, stop/0, elf/0, reindeer/0]).
-export([elf_done/0, reindeer_done/0, status/1]).
-export([init_joins/1]).

start() ->
    jerlang_gen_joins:start({global, ?MODULE}, ?MODULE, [], []).

stop() ->
    jerlang_gen_joins:call({global, ?MODULE}, stop).

terminate() ->
    jerlang_gen_joins:call({global, ?MODULE}, stop).

init(_Args) ->
    {ok, {{0,0}, sleeping}}.

elf() ->
    jerlang_gen_joins:call({global, ?MODULE}, elf),
    ok.

elf_done() ->
    jerlang_gen_joins:cast({global, ?MODULE}, {done, elf}).

reindeer() ->
    jerlang_gen_joins:call({global, ?MODULE}, reindeer).

reindeer_done() ->
    jerlang_gen_joins:cast({global, ?MODULE}, {done, reindeer}).

status(Status) ->
    jerlang_gen_joins:call({global, ?MODULE}, {status, Status}, infinity).

init_joins(_) ->
    {ok,
     [
      [{1, #pattern_joins{test=fun(Value) ->
                    case Value of
                        {status, _, _} ->
                        true;
                        _ ->
                        false
                    end
                end, msgs=[]}},
       {fun([elf, elf],
        {_, sleeping}) ->
        true
       end, []},
       {fun([{status, A, B}],
        [{C, D}, _]) when ((A =< C) and (B =< D)) ->
        true
       end, []]},
      [{1, #pattern_joins{test=fun(Value) ->
                    case Value of
                        stop ->
                        true;
                        _ ->
                        false
                    end
                end, msgs=[]}},
       {fun([stop],
        _) ->
        true
       end, []]},
      [{1, #pattern_joins{test=fun(Value) ->
                    case Value of
                        {done, reindeer} ->
                        true;
                        _ ->
                        false
                    end
                end, msgs=[]}},
```



```
    {6, #pattern_joins{test=fun(Value) ->
                    case Value of
                        {done, reindeer} ->
                        true;
                        _ ->
                        false
                    end
                end, msgs=[]}},
    {7, #pattern_joins{test=fun(Value) ->
                    case Value of
                        {done, reindeer} ->
                        true;
                        _ ->
                        false
                    end
                end, msgs=[]}},
    {8, #pattern_joins{test=fun(Value) ->
                    case Value of
                        {done, reindeer} ->
                        true;
                        _ ->
                        false
                    end
                end, msgs=[]}},
    {9, #pattern_joins{test=fun(Value) ->
                    case Value of
                        {done, reindeer} ->
                        true;
                        _ ->
                        false
                    end
                end, msgs=[]}},
       {fun([{done, reindeer}, {done, reindeer}],
        [_, awake_reindeer]) ->
        true
    end, []},
       {fun([{done, reindeer}, {done, reindeer}, {done, reindeer}],
        [_, awake_reindeer]) ->
        true
    end, []},
       {fun([{done, reindeer}, {done, reindeer}, {done, reindeer},
         {done, reindeer}],
        [_, awake_reindeer]) ->
        true
    end, []},
       {fun([{done, reindeer}, {done, reindeer}, {done, reindeer},
         {done, reindeer}, {done, reindeer}],
        [_, awake_reindeer]) ->
        true
    end, []},
       {fun([{done, reindeer}, {done, reindeer}, {done, reindeer},
         {done, reindeer}, {done, reindeer}, {done, reindeer}],
        [_, awake_reindeer]) ->
        true
    end, []},
       {fun([{done, reindeer}, {done, reindeer}, {done, reindeer},
         {done, reindeer}, {done, reindeer}, {done, reindeer},
         {done, reindeer}],
        [_, awake_reindeer]) ->
        true
    end, []},
       {fun([{done, reindeer}, {done, reindeer}, {done, reindeer},
         {done, reindeer}, {done, reindeer}, {done, reindeer},
         {done, reindeer}, {done, reindeer}],
        [_, awake_reindeer]) ->
        true
    end, []},
      [{1, #pattern_joins{test=fun(Value) ->
                    case Value of
                        {done, elf} ->
                        true;
                        _ ->
                        false
                    end
                end, msgs=[]}},
    {2, #pattern_joins{test=fun(Value) ->
                    case Value of
                        {done, elf} ->
                        true;
                        _ ->
                        false
                    end
                end, msgs=[]}},
    {3, #pattern_joins{test=fun(Value) ->
                    case Value of
```



```
                    case Value of
                        reindeer ->
                        true;
                        _ ->
                        false
                    end
                end, msgs=[]}},
    {5, #pattern_joins{test=fun(Value) ->
                    case Value of
                        reindeer ->
                        true;
                        _ ->
                        false
                    end
                end, msgs=[]}},
    {6, #pattern_joins{test=fun(Value) ->
                    case Value of
                        reindeer ->
                        true;
                        _ ->
                        false
                    end
                end, msgs=[]}},
    {7, #pattern_joins{test=fun(Value) ->
                    case Value of
                        reindeer ->
                        true;
                        _ ->
                        false
                    end
                end, msgs=[]}},
    {8, #pattern_joins{test=fun(Value) ->
                    case Value of
                        reindeer ->
                        true;
                        _ ->
                        false
                    end
                end, msgs=[]}},
    {9, #pattern_joins{test=fun(Value) ->
                    case Value of
                        reindeer ->
                        true;
                        _ ->
                        false
                    end
                end, msgs=[]}},
       {fun([reindeer, reindeer],
        {_, sleeping}) ->
        true
    end, []},
       {fun([reindeer, reindeer, reindeer],
        {_, sleeping}) ->
        true
    end, []},
       {fun([reindeer, reindeer, reindeer,
         reindeer],
        {_, sleeping}) ->
        true
    end, []},
       {fun([reindeer, reindeer, reindeer,
         reindeer, reindeer],
        {_, sleeping}) ->
        true
    end, []},
       {fun([reindeer, reindeer, reindeer,
         reindeer, reindeer, reindeer],
        {_, sleeping}) ->
        true
    end, []},
       {fun([reindeer, reindeer, reindeer,
         reindeer, reindeer, reindeer,
         reindeer],
        {_, sleeping}) ->
        true
    end, []},
       {fun([reindeer, reindeer, reindeer,
         reindeer, reindeer, reindeer,
         reindeer, reindeer],
        {_, sleeping}) ->
        true
    end, []},
```

Santa Claus (**gen_joins**) with parse_transform

67 lines of readable code

```
-module(jerlang_gen_joins_test_santa_parse).
-compile({parse_transform, jerlang_gen_joins_parse}).

-behaviour(jerlang_gen_joins).

-export([init/1, handle_join/2, terminate/0]).
-export([start/0, stop/0, elf/0, reindeer/0]).
-export([elf_done/0, reindeer_done/0, status/1]).

start() ->
    jerlang_gen_joins:start({global, ?MODULE}, ?MODULE, [], []).

stop() ->
    jerlang_gen_joins:call({global, ?MODULE}, stop).

terminate() ->
    ok.

init(_) ->
    {ok, {{0,0}, sleeping}}.

elf() ->
    jerlang_gen_joins:call({global, ?MODULE}, elf, infinity),
    ok.

elf_done() ->
    jerlang_gen_joins:cast({global, ?MODULE}, {done, elf}).

reindeer() ->
    jerlang_gen_joins:call({global, ?MODULE}, reindeer, infinity).

reindeer_done() ->
    jerlang_gen_joins:cast({global, ?MODULE}, {done, reindeer}).

status(Status) ->
    jerlang_gen_joins:call({global, ?MODULE}, {status, Status}, infinity).

handle_join({status, {A, B}}, {{C, D}, _}=S) when ((A =< C) and (B =< D))
->
    io:format("I am done ~p~n", [{C, D}]),
    {[{reply, ok}], S};
handle_join(stop, _) ->
    io:format("Stopping the server~n", []),
    {stop, normal};
handle_join({done, reindeer} and {done, reindeer} and {done, reindeer} and
        {done, reindeer} and {done, reindeer} and {done, reindeer} and
        {done, reindeer} and {done, reindeer} and {done, reindeer},
        {Counter, awake_reindeer}) ->
    io:format("All reindeers returned~n",[]),
    {[ noreply || _ <- lists:seq(1,9)], {Counter, sleeping}};

handle_join({done, elf} and {done, elf} and {done, elf},
        {Counter, awake_elf}) ->
    io:format("All elves returned~n",[]),
    {[ noreply || _ <- lists:seq(1,3)], {Counter, sleeping}};

handle_join(reindeer and reindeer and reindeer and
        reindeer and reindeer and reindeer and
        reindeer and reindeer and reindeer,
        {{Reindeers, Elves}, sleeping}) ->
    io:format("Ho, ho, ho! Let's deliver presents~n",[]),
```

VM or non-VM

Self-contained JErlang library:

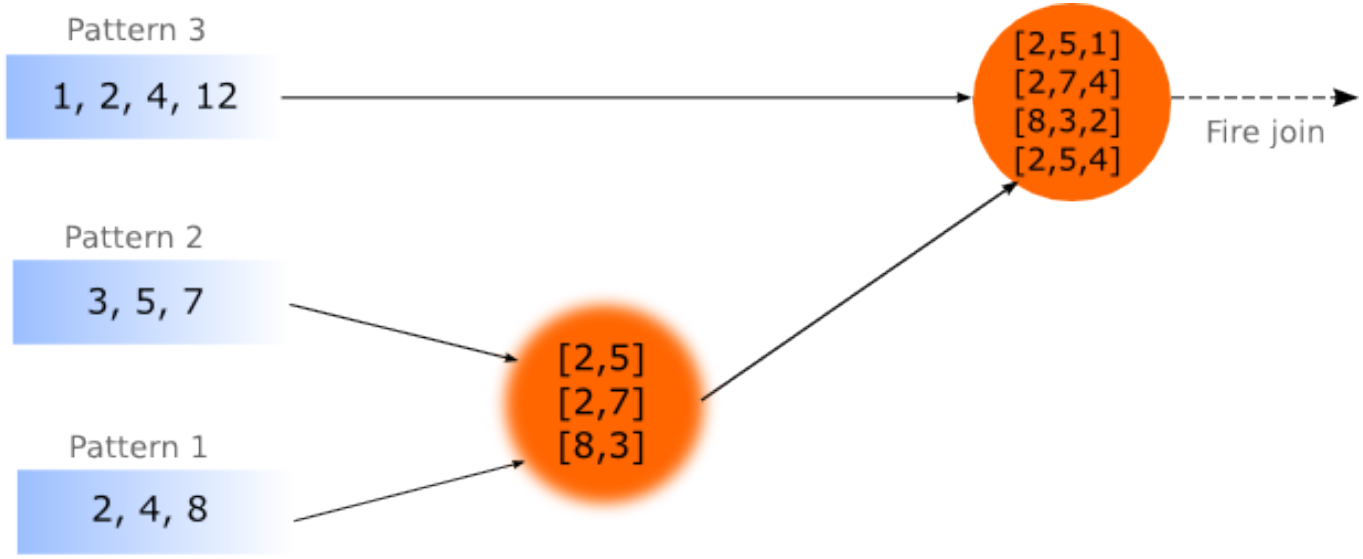
- Portable
- Requires only a single line in the original code
- Introduces overhead related to mailbox access/storage (library mailbox with processed messages and the unprocessed ones in the VM)

VM-accelerated JErlang library:

- Implemented without any documentation of VM or compiler (C, Erlang)
- Requires patched version of the Erlang compiler and run-time
- Provides primitives necessary to efficiently access the mailbox
- Uses modified version of the self-contained library
- Significantly faster in many situations
- Uses hash-map data structure to complement VM's queue

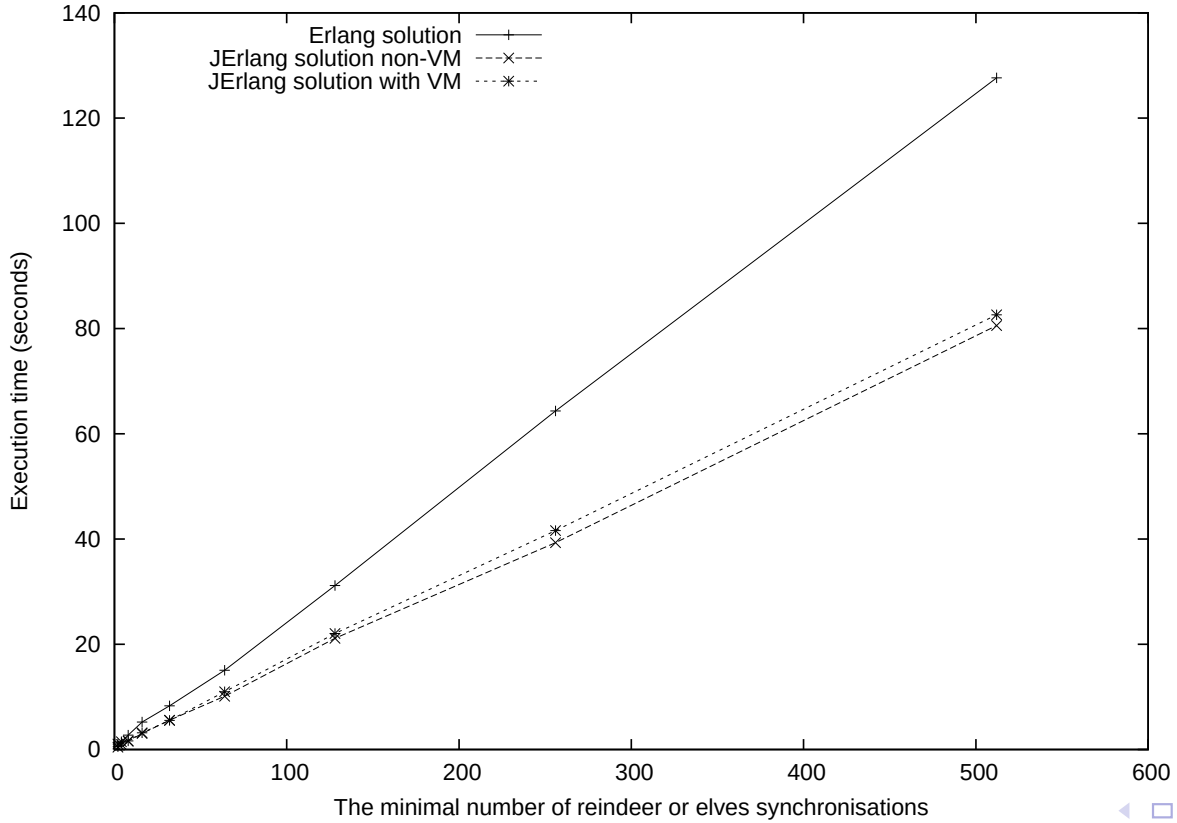
RETE algorithm

- Well used in Production Rule Systems for more than 30 years
- Initially applicable to **gen_joins** implementation, used in VM and non-VM implementation
- Avoids re-computation of results (better performance for larger mailboxes and complicated patterns)



Performance

- Roughly similar when comparing respective language features in other implementations
- Suffers from "too-much-mail is bad for you" problem
- Various optimisation techniques used



Further optimisations

- Joins' patterns ordering

```
1 receive
2     {foo, One} and test1 and test2 and {bar, One} ->
3         ...
4 end
```

vs

```
1 receive
2     {foo, One} and {bar, One} and test1 and test2 ->
3         ...
4 end
```

- Early application of variables for faster message filtering
- Detection of identical messages

Future work

- Parallel joins solver (constraints on the expressiveness)
- Various static analysis of the code patterns to boost performance
- More efficient VM-implementation (data structures)
- Formalisation of the equivalence of Erlang and JErlang
- Go open-source (roughly 7 000 lines of code without VM changes)

Achievements

- Defined formal syntax and semantics of the extension
- Implemented extension of Erlang with consistent semantics that increases the expresiveness of the language and provides features that do not exist in other joins implementations
- Implemented self-standing JErlang library with and without VM/compiler changes
- Used static analysis techniques for producing better code
- Investigated various optimisation techniques that significantly increase the usability of joins

Santa Claus problem - Polyphonic C# (C ω)

Elegant? Intuitive?

```
1  public class nway {
2   public async produce(int n) & public void consume() {
3       if (n==1) {
4           alldone();
5       } else {
6           produce(n-1);
7       }
8   }
9
10  public void waitforalldone() & async alldone() {
11      return;
12  }
13 }
```

Santa Claus problem - Polyphonic C# part 2

```
1  class santa {
2   static nway harness = new nway();
3   static nway unharness = new nway();
4   static nway roomin = new nway();
5   static nway roomout = new nway();
6
7   static void santalife() {
8       while (true) {
9           waittobewoken();
10          // get back here after dealing with elves or reindeer
11      }
12  }
13
14  static void waittobewoken() & static async elvesready() {
15      roomin.produce(3);
16      roomin.waitforalldone();
17      elveswaiting(0);
18      // all elves in the room, consult
19      roomout.produce(3);
20      roomout.waitforalldone();
21      // all elves shown out, go back to bed
22  }
23
24  static void waittobewoken() & static async reindeerready() {
25     // similar to elvesready chord
26  }
```

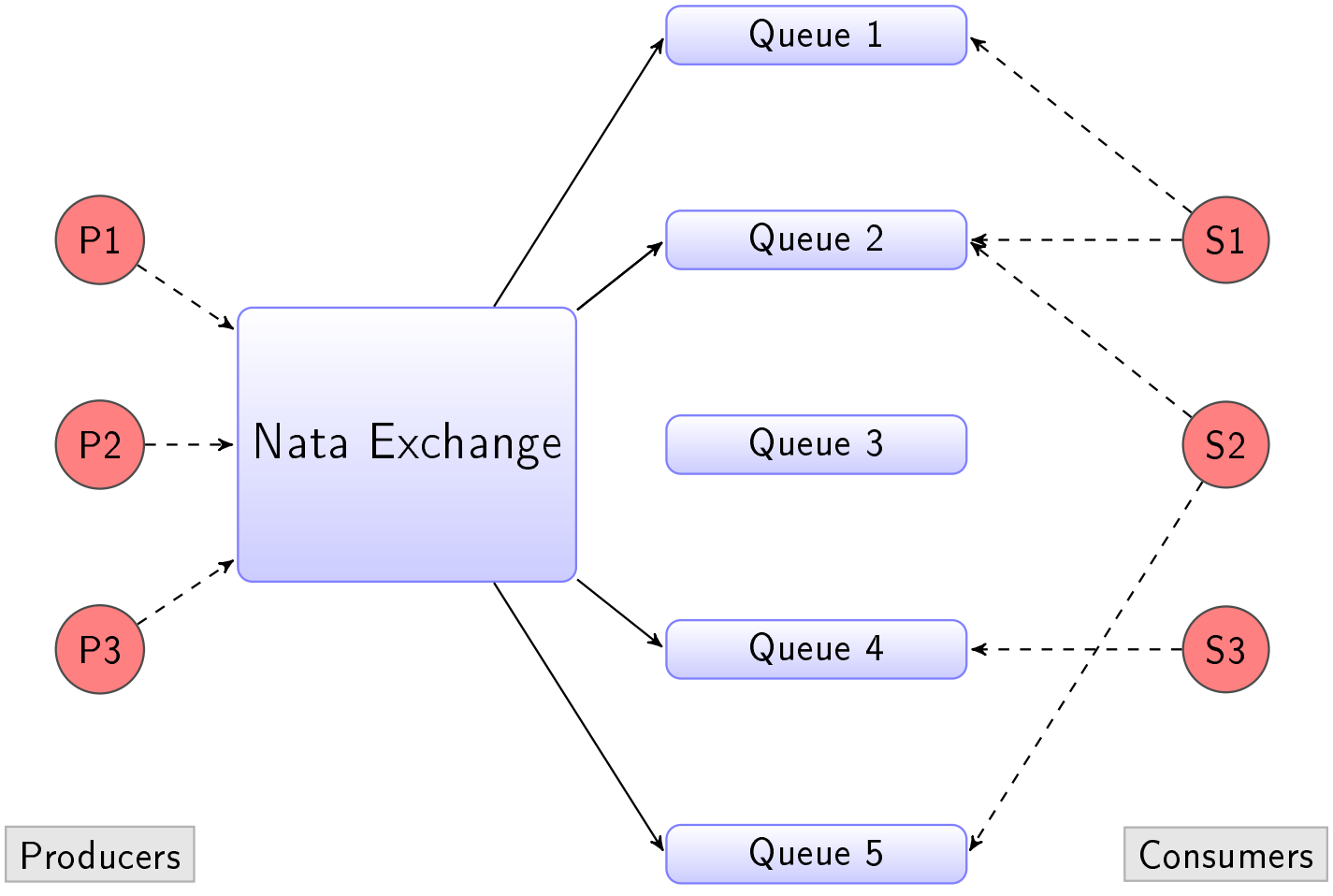
Santa Claus problem - Polyphonic C# part 3

```
1  static async elflife(int elfid) {
2    while (true) {
3     // work
4     elfqueue();          // wait to join group of 3
5     roomin.consume();    // wait to be shown in
6     // consult with santa
7     roomout.consume(); // wait to be shown out again
8    }
9   }
10
11  static void elfqueue() & static async elveswaiting(int e) {
12    if (e==2) {
13      elvesready();       // wake up santa
14    }
15    else {
16      elveswaiting(e+1);
17   }
18  }
```

Expandable? No prioritisation - requires further, hard to maintain, “hacks”.

JErlang allows for much more intuitive solution of the problem.

NataMQ


Queue 1
Queue 2
Queue 3
Queue 4
Queue 5
P1
P2
P3
Nata Exchange
S1
S2
S3
Producers
Consumers