

# Priority Messaging made Easy

## A Selective Generic Server

Jan Henry Nyström

Erlang Training and Consulting Ltd.  
jan@erlang-consulting.com

### Abstract

This paper provides an introduction to the finer points of priority based message reception. A new behaviour is devised to provide a generic server that does priority based message reception.

The combination of generic finite state machines and prioritised is also discussed.

Finally it is demonstrated how behaviours could be significantly strengthened in usefulness by allowing the behaviour info to specify that a parse transform. An Erlang Extension Proposal suggesting the change is included in the appendices.

**Categories and Subject Descriptors** D.2.3 [Coding Tools and Techniques]: Structured Programming; D.3.3 [Language Constructs and Features]: Input/Output

**General Terms** Languages, Measurement, Performance

**Keywords** Erlang, Generic, Server, Behaviour

### 1. Introduction

The paper addresses the two major shortcomings with prioritised reception of messages in ERLANG .

The first shortcoming is that prioritised messages reception is often perceived as complicated and error prone, once you have more than one priority level. In fact it quite often comes as a nasty surprise, that if you need as strict prioritisation as possible you have to have an one element look-ahead.

The second shortcoming is that prioritised messages reception can not be combined in a easy and useful way with with the existing OTP behaviours.

We will address these shortcomings by providing a new behaviour that is a generic server that can prioritise the message selection. This will remove the need to write the complicated part of the prioritisation as well as having all the advantages of an OTP behaviour.

We will develop the new behaviour by going through some of the possible design options, listing the pros and cons for each, before settling for a design. The set of design decisions are motivated. But first we will describe priority messaging in more details, pointing out some of the associated problems.

The paper will then go on to advocate a significant change in the behaviours, that allows more powerful behaviours. The change is conservative in the sense that it does not break any existing code.

### 2. Priority Messaging Explained

The principle behind priority reception of messages is very simple; normally each message in the queue is sequentially checked against all the clauses in the `receive` statement, and then if no message matches any of the clauses the call blocks. But the `receive` statement comes with an optional timeout clause, the body of which is evaluated if no matching message is received within the specified time. If the timeout time is given as 0, the call never blocks.

In order to give a specific message reception priority, we do a non-blocking (with timeout 0) `receive` that has as the timeout clause body the normal reception. This is all well understood and presented in Armstrong et al. [1996].

A small example can be seen in Figure 1. where any message included in a two-tuple with the first element the atom `priority` will be received, if present in the mailbox, before any other message.

In Armstrong [2007] the new ERLANG standard reference it is noted that:

Using large mailboxes with priority receive is rather inefficient, so if you're going to use this technique, make sure your mailboxes are not too large.

This is a very important point, that merits an extra highlighting. However the story does not end here, for one thing we have only dealt with one level of priorities.

```
prio() ->
receive
  {priority, X} -> X
after 0 ->
  receive
    X -> X
  end
end.
```

**Figure 1.** A small priority reception of messages example.

#### 2.1 Pitfalls and Penalties

The main problem with the example in Figure 1, is that we do not take into consideration that when evaluation is resumed from the inner blocking `receive` we may have more than one message in the mailbox. In a worst case scenario, all but the first, of potentially a huge number, of elements could be priority messages. In this scenario we would actually have accomplished the very opposite of what we intended to do.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

Erlang'07, October 5, 2007, Freiburg, Germany.

Copyright © 2007 ACM 978-1-59593-675-2/07/0010...\$5.00

Another problem with that example is that we only have one priority level. We could have any number of priority levels. The coding principle extends easily to two levels as in Figure 2 and so on. The problem is just that our performance penalty just keeps going up. For each priority level we will in the worst case traverse the entire mailbox, and that for each message received.

So on the whole, the complexity would be: (the number of priority levels) times (the number of clauses in each priority level) times (the number of messages in the mailbox). The situation gets even worse with a look-ahead.

**Priority messaging golden rule 1:** Only use priority message reception if you really need it. There will always be a penalty to pay.

**Priority messaging golden rule 2:** If you use priority message reception, you must ensure that you never have too many elements in the mailbox. This is achieved by limiting the number of messages sent to the process.

**Priority messaging golden rule 3:** Do not use more priority levels than absolutely necessary.

```
prio() ->
receive
  {priority1, X} -> X
after 0 ->
  receive
    {priority2, X} -> X
  after 0 ->
    receive
      X -> X
    end
  end
end
end
```

Figure 2. A two priorities example.

## 2.2 The Right Way™

The solution with the blocking call picking the first element is to see whether the received message was in fact a priority message, and if not first ensure that we do not have any in the mailbox before proceeding.

An example of this is presented in Figure 3. Now this solution suffers from a small weakness, it discards the non-priority messages in the function `priop/1`. Most of the time this is not a luxury we can afford, which means we have to keep track of the unprioritised look-ahead value we have received.

```
prio() ->          priop(Peek) ->
  receive          case Peek of
    {priority, X} -> X  {priority, X} -> X;
  after 0 ->          - ->
    receive          receive
      X -> priop(X)    {priority, X} -> X
    end              after 0 -> Peek
  end.              end
end.                end.
```

Figure 3. Priority reception of messages with look-ahead.

To keep track of the look-ahead we have two choices, first we can work the look-ahead into the fabric of the code via a field in the process state (passed from recursive call to recursive call in the main loop). Secondly we can use the process dictionary.

Normally the advice is to do the first, but to get a self-contained example, one that ties well in with the actual implementation, a working example with a look-ahead saved in the process dictionary is shown in Figure 4. In the example we either have a look-ahead and it is a priority message, or we do not have a look-ahead, or we have one that is not a priority message.

```
prio() ->
  Peek = erlang:get(peek)
  case Peek of
    {{priority, X}} ->
      erlang:put(peek, undefined),
      X;
    undefined ->
      receive
        {priority, X} -> X
      after 0 ->
        receive
          X ->
            erlang:put(peek, {Peek}),
            prio()
          end;
        - ->
          receive
            {priority, X} ->
              erlang:put(peek, {Peek}),
              X
            after 0 ->
              erlang:put(peek, undefined),
              Peek
            end
          end
        end
      end.
```

Figure 4. Priority reception of messages with ERLANG dictionary stored look-ahead.

To make matters even more complicated we should add several priority levels to the example in Figure 4. Then in the case when we have a look-ahead that is not a first level priority message we have to check if it is a second level priority message and so on. You are spared the gory details for now, let it suffice to say that “Yes priority message reception is complicated and therefore error prone”. These details are presented in the generated functions in Section 4.1.

## 3. Implementation Choices and Trade-offs

The solution to our problems will be a behaviour that provides a generic server with priority message reception at the cost, for the user, of one compiler attribute. But first we have to decide how to construct this wonder. We will go through a number of design issues and for each list the pros and cons before presenting a complete set of design decisions.

### 3.1 Cascading Behaviours

The solution that springs to mind is to extend the existing generic server with another abstraction layer, constructing a so called cascading behaviour. The generic part of the new behaviour would constitute the call back module for generic server.

This however does not work since generic server does all the reception in order and the callbacks are only called on reception. The only means of detecting that we have no further incoming messages exists are timeouts. Consequently there is no feasible way to extend then generic server this way.

### 3.2 That New Behaviour Smell

Another appealing approach is always to construct a brand new behaviour from scratch using the `gen` library. This has the advantage that a small server with a minimum of functionality can be constructed.

This approach also provides the greatest flexibility when it comes to the other design choices we have to make. It is furthermore not dependant on another behaviour.

What speaks against it, is that the functionality provided by the present generic server is there as a result of a selection based on many years experience dealing with concurrent and distributed applications. Any generic server is probably bound to include them in the end, so why not use a tried and tested implementation that already exists and is supported.

### 3.3 Patch and Go

The final approach would be to make a minimal, and as self-contained as possible, change to the existing generic server. Splicing in the needed functionality. The construction of the new behaviours would be the application of a patch to the existing behaviour.

This approach would benefit from the existing implementation, a well tried and tested code base. But it could also profit from correction of the existing generic server.

On the other hand possible futures changes might make the splicing more difficult and every time a new version is released it would have to be regression tested.

Another important factor is that for the users the existing generic servers are something old and familiar, the learning curve for the new behaviour is on the verge of being a flat line. What more is that the documentation needed apart from the existing is very small.

### 3.4 Declared or Automatically Derived

The purpose of all this is to provide priority messages reception but we have not yet addressed how we determine what messages should be prioritised. We can either infer that from the clauses dealing with the different messages or we would have to declare which messages have priority.

To infer the priorities would mean that we only can have as many function clauses for this as there are priority levels, and in most cases this more than one level would be expensive.

This would lead to a situation where we only have two clauses: one for the priority messages and one for the rest. This hardly leads to clear and concise code. One of the general rule for readable ERLANG code is to have as much selection at the function clause level.

Explicitly giving a declaration of the priorities makes for much clearer reading with priorities collected and exposed. Furthermore the use of priorities can be changed by commenting out a single declaration. The downside of centralisation of priority information apart from the clauses, is that it splits messages information across the module.

Explicit declaration can either be in the form of a function returning the information or as a compiler attribute. The attribute will have the advantage that is clear that it is not a function which is intended for use during run-time.

On the other hand a function that is exported could potentially be used to query a process how it prioritises. The author can not see any immediate use of that functionality.

### 3.5 Is there any real difference between Call, Cast and Info

The question is partially if there would be any reason to differentiate between the different types of messages when dealing with priorities; and partially what is most efficient as we would like to minimise the penalty we incur when using priorities.

It is questionable if messages of the same shape, but different type, would be prioritised in the same way. The only reason the author can perceive to organise it that way would be to cut down on the levels of priorities.

An important aspect is whether we base our implementation on the existing generic server since that has very different shape on cast and calls. All other messages, or info in the lingo of generic server, are of course of an unspecified shape. That means, that if when a message is declared as having a priority all casts, calls and info has that priority, we have to generate three clauses for that.

### 3.6 And the winner is?

The clear winner is the patched generic server with explicitly declared priorities as compiler attributes with calls, casts and info messages distinguished. For the declared attributes a compromise is used as a pattern can either be of type `cast`, `call`, `info` or `all`.

Taking all into account the author has come to the conclusion that the benefits of reusing the generic server outweigh the drawbacks. The by far most important factor was the behaviour will be more readily accepted by the users when similar to the existing generic server.

Declared priorities seemed to be the one that would give the clearest code and for the author that makes it no competition since when coding "Clarity is King!"

The choice for distinguishing between the different types of messages it was more touch and go but in the end the potential for more efficiency ruled the day. It is also possible that it will make it more clear and inclusion in prioritisation by mistake less likely.

## 4. Patch and Go with Parse Transforms

When we have reached this point we have actually done all the really hard work. All that remains is to change the `generic_server.erl` module splicing in our prioritisation and export the `parse_transform/2` function and implement it. The resulting behaviour is called `gen_select_server`.

The splicing in is quite straight forward, we only have to change the reception of messages in the main loop (of course a tail recursive function) of the generic server. The lines:

```
Msg = receive
    Input ->
        Input
    after Time ->
        timeout
end,
```

are changed into:

```
Msg = Mod:select(Parent, Time),
```

Where `Mod` is the callback module since the information of what is prioritised resides in the callback module and there is where the generated function will reside.

### 4.1 Parse Transforms

The `parse_transform/2` function is the one that performs the actual magic that makes it all work. In order to understand where the `parse_transform/2` comes into this one has to understand something of how the compiler works.

The compiler first scans the ERLANG module producing a number of tokens signifying syntactical elements such as keywords, atoms, punctuation, etc. Then the tokens are parsed into an abstract parse tree. That tree is transformed to perform various optimisations and checked for various properties<sup>1</sup>. When all the changes

<sup>1</sup>The presentation of the compiler heavily simplified, but adequate for the purposes of this paper.

has been performed the tree is used to generate the byte code or native code.

Parse transforms are performed on the abstract syntax tree that is generated when an ERLANG module is parsed before any of the optimisation passes are called. This enables us to change the code in the callback module.

The `parse_transform/2` function will search the abstract syntax tree for the compiler attribute `selection`, if it exists a selection functions is generated mirroring how we constructed a selection function `prio/0` in Section 2.2. In fact for efficiency and clarity reasons we will generate four functions.

The reason we save the look-ahead in the process dictionary is that we otherwise would have to change not 6 lines but something in the order of 50 lines scattered across the entire module.

The selection attribute will look like:

```
-selection(Selection)
```

where `Selection` is a list of either selection tuples or lists of selection tuples. A selection tuple is a two-tuple where the first element is one of `cast`, `call`, `info` and `all` and the second any bound ERLANG term.

Any atoms with names consisting of a dollar sign followed by an integer are considered variables, e.g., `$1`, `$2`, `$57`. Two 'variables' with the same number are considered to be equal and for match-all either a singleton variable is used or the atom `'_'`. Look in Figure 7 for an example.

Examples of the how the functions would look given the declaration in Figure 5 are found in Figure 10 to Figure 13. The functions have slightly more readable names, as has the name of the look-ahead. The real names are much less esthetically pleasing to avoid accidental name clashes.

The generated selection functions always have an extra priority level, that is because it should act like a proper behaviour. The messages on the form `{system, _, _}` are system messages generated from the `sys` module. They are used, e.g., to suspend process when a code upgrade is performed. The exit messages from the parent must be observed to fulfil the `shutdown` protocol that is followed by all behaviours.

In total the `parse_transform/2` function with auxiliary functions makes circa 250 lines of code.

```
-selection([[{Type.1.1, Pattern.1.1},
            {Type.1.2, Pattern.1.2},
            ...
            {Type.1.n1, Pattern.1.n1}],
          ...
          [{Type.m.1, Patter.m.1},
            ...
            {Type.m.nm, Patter.m.nm}]])).
```

Figure 5. General form of declaration.

## 4.2 Invoking the Parse Transform

In all this we have failed to mention how we invoke the parse transform. This was initially achieved by a compiler directive in the callback module:

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

This instructs the compiler to include the `parse_transform/2` function amongst those applied to the module.

This however means, that not only do we have the behaviour declaration to say what behaviour were writing a callback module for, and all that generates are checks that the right call functions are exported, but we also have to have a separate declaration of the parse transform.

The natural thing, would be for the behaviour declaration to suffice. If the behaviour defines a parse transform this will be run at compile time.

To make this work we have make some small changes to the compiler. The compiler will if there is a behaviour compiler attribute check the `behaviour_info/1` if a parse transform exists. If one exists, that is included among the parse transforms run.

In the behaviour implementation all that has to be added is the clause:

```
behaviour_info(parse_transform) -> true;
```

The changes made to the compiler are detailed in the Erlang Extension Proposal included in Appendix A.

## 5. A Selective Server Example

In this section we showcase a small sample server. The code is found below in the Figure 7.

### 5.1 The Server

The server registers itself when created and the `init/1` callback does nothing, returning an empty state in the form of the atom `none`.

The server will, once it is started, accept all messages, printing information on the shell regarding what kind of message it has received.

It has three interface functions which are used to send the messages, sending a `cast` and `info` message are performed in the usual manner. The `call` is performed by process spawned for that purpose, this is in order to enable a testing process to generate several calls without blocking.

The server also comes with its own test function, that will generate a number of messages that highlights the servers prioritisation of messages.

Finally we have the selection attribute that will make three messages patterns priority, while not making any difference between two first.

### 5.2 A Run in Unprioritised Messages

```
1> test_select_server:start_link().
{ok,<0.32.0>}
2> test_select_server:test().
Cast: [duu]
<0.37.0>Cast: [nepp]
Cast: [third]
Info: [third]
Cast: [{first,1,two}]
Info: [{first,1,two}]
Cast: [{second,a,a}]
Info: [{second,a,a}]
Info: [{first,o,o}]
Cast: [{first,o,o}]
Call: [third]
Call: [{first,1,two}]
Call: [{second,a,a}]
Call: [{first,o,o}]
3>
```

### 5.3 A Run in Prioritised Messages

```
1> test_select_server:start_link().
{ok,<0.32.0>}
2> test_select_server:test().
Cast: [nepp]
<0.37.0>Call: [{first,o,o}]
Cast: [{second,a,a}]
```

```

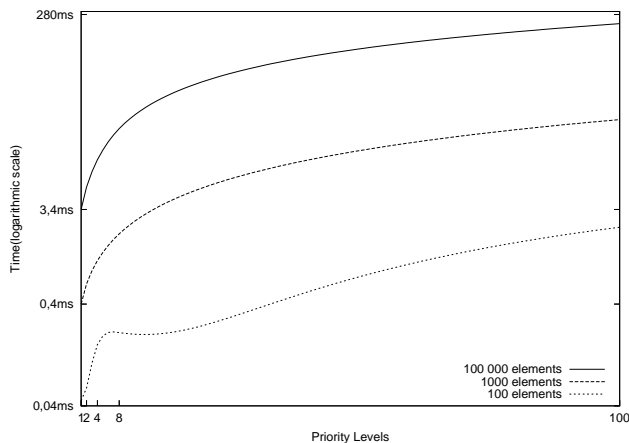
Cast: [duu]
Cast: [third]
Info: [third]
Cast: [{first,1,two}]
Info: [{first,1,two}]
Info: [{second,a,a}]
Info: [{first,o,o}]
Cast: [{first,o,o}]
Call: [third]
Call: [{first,1,two}]
Call: [{second,a,a}]
3>

```

## 6. The Price of Priorities

We have several times come back to the cost of using prioritised reception of messages, but what are the actual costs? We have performed a simple benchmark, intended to give a rough idea of the costs involved.

The benchmark consists of running our test server from Section 5.1 with a different set of selection attributes and measuring the time it takes to pick an element out a message queue where no element in it is prioritised (the worst case).



The time is given in logarithmic scale, showing that the times are proportional to the square of the number of priority levels. Each priority level only has one element.

**Figure 6.** Time to retrieve one message from the queue.

We repeat the experiment several times with a varying number of elements in the queue, levels of priorities and elements of the priority levels. The queue lengths are 1, 2, 4, 8, 100, 1000 and 10000. The number of levels 1, 2, 4, 8, 100 and finally the number of elements per level 1, 2, 4, 8, 100 (except for 100 priority case levels where only one element per level is used).

The benchmark was performed on a 1GHz Pentium III with 1Gb of memory. On the test machine a normal receive takes roughly 0,0005ms.

The detail results are presented in Figures 6 to 9. The cheapest setting is one priority level with one element and one element in the queue with 0,015ms. The dearest tested is with 100 priority levels and 10 000 messages in the queue taking 228ms.

## 7. What about Finite Machines then?

Over the years the author has heard as many requests for prioritised reception of events for generic finite state machines as for that of

```

-module(test_select_server).
-behaviour(gen_select_server).

-export([start_link/0, call/1,
         cast/1, info/1, test/0]).

-export([init/1,
         handle_call/3,
         handle_cast/2,
         handle_info/2,
         terminate/2,
         code_change/3
        ]).

-selection([{{call, {first, '$1', '$1'}},
           {all, nepp}},
          {cast, {second, '$1', '$1'}}]).

start_link() ->
  gen_select_server:start_link({local, ?MODULE},
                               ?MODULE,
                               no_args,
                               []).

call(Msg) ->
  spawn(gen_select_server, call, [?MODULE, Msg]).
cast(Msg) -> gen_select_server:cast(?MODULE, Msg).
info(Msg) -> ?MODULE ! Msg.
init(no_args) -> {ok, none}.

handle_call(Msg, _, none) ->
  io:format("Call: [~p]~n", [Msg]),
  {reply, ok, none}.

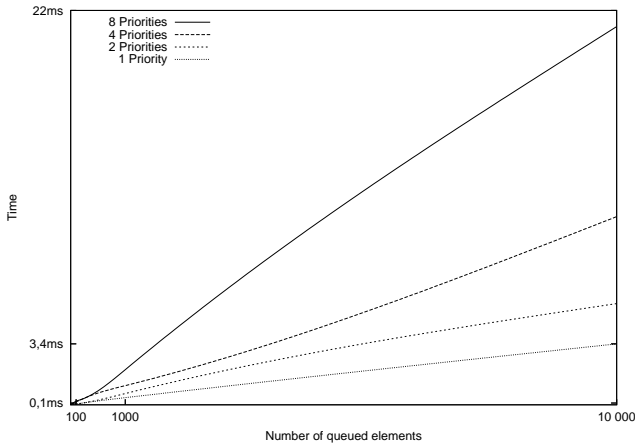
handle_cast(Msg, none) ->
  io:format("Cast: [~p]~n", [Msg]),
  {noreply, none}.

handle_info(Msg, none) ->
  io:format("Info: [~p]~n", [Msg]),
  {noreply, none}.

test() ->
  cast(duu), cast(nepp),
  call(third), cast(third), info(third),
  call({first, 1, 2}), cast({first, 1, 2}),
  info({first, 1, 2}),
  cast({second, a, a}), call({second, a, a}),
  info({second, a, a}),
  info({first, o, o}), cast({first, o, o}),
  call({first, o, o}).

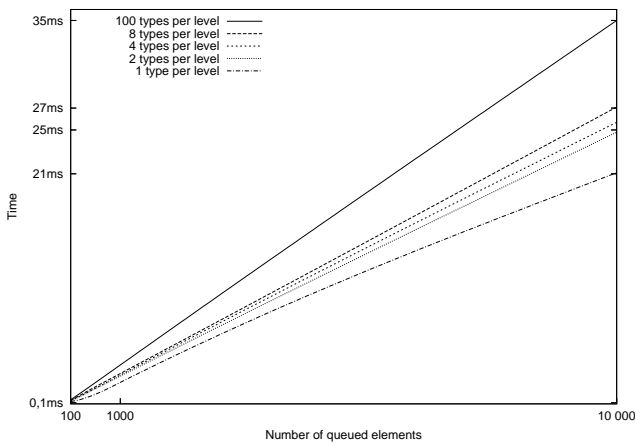
```

**Figure 7.** A small selective generic server.



The same information as in Figure 6 but with y-axis being the time and the x-axis the number of queued elements. The case with 100 priority levels is excluded to enhance readability.

**Figure 8.** Time to retrieve one message from the queue.



The time to retrieve one message with different number of elements per priority level, with 8 priority levels.

**Figure 9.** Time to retrieve one message from the queue.

messages for generic servers. The generic servers were selected as they would make a clearer example.

Most of what has been presented in this paper is valid for FSMs as well. The main difference are the states which complicates the issue somewhat. Does one make difference between states as well as types of events for priorities?

The nature of events would dictate that one would treat priorities differently for states for normal events, but not for 'send all events' and info since they deal with out of the normal events that in most cases are dealt with similarly for all states.

The generation of the selection functions will be slightly more complicated and generate more code, but apart from that following the steps presented above for generic servers should work a treat.

In fact the author has an implementation for `gen_select_fsm`, which is planned to be released at the same time as the `gen_select_server` in the "Users Contribution Forum" at [www.trapexit.org](http://www.trapexit.org).

```
select(Parent, Time) ->
receive
  Y = {system,_,_} -> Y;
  Y = {'EXIT',Parent,_} -> Y
after
  0 ->
    receive
      Y = Pattern.1.1 -> Y;
      ...
      Y = Pattern.1.n1 -> Y
    after
      0 ->
        case get(peek) of
          undefined ->
            select1(Parent, Time);
          {Peek} ->
            put(peek, undefined),
            select2(Peek)
        end
      end
    end
end.
```

**Figure 10.** Generated main selection function.

## 8. Conclusions or Did Anyone Say EEP?

This paper has discussed some of the finer points of priority messages reception in ERLANG. It has further shown how one can have generic servers and priorities provided one is prepared to construct a new behaviour extending the existing generic server.

What can be concluded from all this? First that providing new behaviours is nothing magical or even terribly complicated. The most complicated part of the endeavour was the priorities. And that was when one has multiple levels priorities.

Secondly we have seen that adding the ability in behaviours to invoke a parse transform greatly enhances what can be achieved and how we can extend existing behaviours. The author strongly suggest that this ability should be added to the OTP compiler. An Erlang Extension Proposal is found in Appendix A.

## References

- J. Armstrong. *Programming Erlang – Software for a Concurrent World*. Pragmatic Programmer, 2007.
- J. Armstrong, R. Virding, C. Wikström, and M. Williams. *Concurrent Programming in ERLANG*. Prentice Hall, 2nd edition, 1996.

## A. Appendix EEP

EEP: XXX

Title: Adding Parse Transforms to Behaviours

Version: \$Rev\$

Last-Modified: \$Date\$

Author: Jan Henry Nystrom <jan@erlang-consulting.com>

Status: Draft

Type: Standards Track

Content-Type: text/plain

Created: 05-Jul-2007

Erlang-Version: R11-B6

Post-History:

Abstract

The power of behaviours would be greatly improved if they could invoke parse transforms implicitly by adding information in the `behaviour_info/1` call back function in the implementing thus enabling compile time changes.

Rationale

The ability would make the creation of new behaviours easier by allowing skeleton behaviours to be added that, together with a few compiler directives generate new behaviours extending the skeleton (or indeed fully fledged stand alone) behaviours. Thus new behaviours could be constructed with a minimum of coding and sometimes completely bypassing the difficult parts of writing behaviours. For an example see [1].

Changes Necessary

The only part of OTP that needs to be changed is the compiler and linter. In compiler the part of `compile.erl` that reads:

```
compile_options([attribute,_L,compile,C}|Fs]) when is_list(C) ->
  C ++ compile_options(Fs);
compile_options([attribute,_L,compile,C}|Fs]) ->
  [C|compile_options(Fs)];
compile_options([_F|Fs]) -> compile_options(Fs);
compile_options([]) -> [].
```

would be changed to:

```
compile_options([attribute,_L,behavior,C}|Fs]) ->
  behaviour_option(C, Fs);
compile_options([attribute,_L,behaviour,C}|Fs]) ->
  behaviour_option(C, Fs);
compile_options([attribute,_L,compile,C}|Fs]) when is_list(C) ->
  C ++ compile_options(Fs);
compile_options([attribute,_L,compile,C}|Fs]) ->
  [C|compile_options(Fs)];
compile_options([_F|Fs]) -> compile_options(Fs);
compile_options([]) -> [].
```

```
behaviour_option(C, Fs) ->
  case catch C:behaviour_info(parse_transform) of
    true -> [{parse_transform, C}|compile_options(Fs)];
    _ -> compile_options(Fs)
  end.
```

References

[1] Priority Messaging made Easy, Jan Henry Nystrom, Submitted to Erlang Workshop 2007, Freiburg, Germany.

Copyright

This document has been placed in the public domain

```

select1(Parent, Time) ->
  receive
    Y = Pattern.2.1 -> Y;
    ...
    Y = Pattern.2.n2 -> Y
  after
    0 ->
      receive
        Y = Pattern.3.1 -> Y;
        ...
        Y = Pattern.3.n3 -> Y
      after
        0 ->
          ...
          receive Peek ->
            select3(Peek, Parent)
          after Time -> timeout
          end
          ...
        end
      end
    end
  end.

```

---

**Figure 11.** The first generated helper function that without look-ahead

```

select2(Peek) ->
  case Peek of
    Y = Pattern.2.1 -> Y;
    ...
    Y = Pattern.2.n2 -> Y
  - ->
    receive
      Y = Pattern.2.1 ->
        put(peek, {Peek}),
        Y;
      ...
      Y = Pattern.2.n2 ->
        put(peek, {Peek}),
        Y
    after
      0 ->
        case Peek of
          Y = Pattern.3.1 -> Y;
          ...
          Y = Pattern.3.n3 -> Y
        - ->
          receive
            Y = Pattern.2.1 ->
              put(peek, {Peek}),
              Y;
            ...
            Y = Pattern.2.n3 ->
              put(peek, {Peek}),
              Y
          after
            0 ->
              ...
              Peek
              ...
            end
          end
        end
      end
    end
  end.

```

---

**Figure 12.** The second generated helper function with a look-ahead

```

select3(Peek, Parent) ->
case Peek of
  Y = {system,_,_} -> Y;
  Y = {'EXIT',Parent,_} -> Y
- ->
  receive
    Y = {system,_,_} ->
      put(peek, {Peek}),
      Y;
    Y = {'EXIT',Parent,_} ->
      put(peek, {Peek}),
      Y
  after
    0 ->
      case Peek of
        Y = Pattern.1.1 -> Y;
        ...
        Y = Pattern.1..n1 -> Y
      - ->
        receive
          Y = Pattern.1.1 ->
            put(peek, {Peek}),
            Y;
          ...
          Y = Pattern.1..n1 ->
            put(peek, {Peek}),
            Y
        after
          0 ->
            ...
            Peek
            ...
        end
      end
    end
  end
end
end.

```

---

**Figure 13.** The third generated helper function called by function two when resuming after blocking receive.