Erlando

Imitation (of syntax)
is the most sincere form of flattery

Matthew Sackman
matthew@rabbitmq.com

Erlang Syntax: some observations

RabbitMQ

Recent blog post by David Pollak:

> *So, why is Erlang not part of the mainstream? Why aren't developers flocking to it?*
>
> *First, it's ugly. I never really understood how aesthetics impact developers, but beauty on the developer's screen does lead to better results…*

Examples: record syntax; it's a parenthesis-ridden language; numbered variables…

Erlang Syntax: some observations

RabbitMQ

Recent blog post by David Pollak:

> *Fourth, Erlang is all immutable all the time. This is hard to grasp and work with, especially coming from a Java or Ruby background. And without state monads and other pieces that the Haskell folks built into the libraries, the only way to carry state around your application is on the stack via recursion within processes/Actors and message passing…*

Recent blog post by David Pollak:

> *Fourth, Erlang is all immutable all the time. This is hard to grasp and work with, especially coming from a Java or Ruby background. And without state monads and other pieces that the Haskell folks built into the libraries, the only way to carry state around your application is on the stack via recursion within processes/Actors and message passing…*

Immutable: *tough – get used to functional languages*

Recent blog post by David Pollak:

> *Fourth, Erlang is all immutable all the time. This is hard to grasp and work with, especially coming from a Java or Ruby background. And without state monads and other pieces that the Haskell folks built into the libraries, the only way to carry state around your application is on the stack via recursion within processes/Actors and message passing…*

Immutable: *tough – get used to functional languages*
Lack of machinery for abstractions: *we can fix that…*

ERLANDO

- A set of extensions for Erlang
- Implemented as parse-transformers (just like QLC)
- *cut* – implements Scheme-like cuts (cheap syntax for partial application / currying)
- *do* – implements Haskell-like do-notation
- *import-as* – very simple remote function importing with aliasing

```
info_all(VHostPath, Items) ->
  map(VHostPath, fun (Q) -> info(Q, Items) end).
```

```
info_all(VHostPath, Items) ->
  map(VHostPath, fun (Q) -> info(Q, Items) end).

backing_queue_timeout(State = #q { backing_queue = BQ }) ->
  run_backing_queue(
    BQ, fun (M, BQS) -> M:timeout(BQS) end, State).
```

```
info_all(VHostPath, Items) ->
  map(VHostPath, fun (Q) -> info(Q, Items) end).

backing_queue_timeout(State = #q { backing_queue = BQ }) ->
  run_backing_queue(
    BQ, fun (M, BQS) -> M:timeout(BQS) end, State).

reset_msg_expiry_fun(TTL) ->
  fun (MsgProps) ->
    MsgProps #message_properties {
      expiry = calculate_msg_expiry(TTL) }
  end.
```

- All three cases show funs being created to supply parameters to simple expressions.

- All three cases show funs being created to supply parameters to simple expressions.
- Really, this is partial application: some parameters to the expressions won't be known until later on.

- All three cases show funs being created to supply parameters to simple expressions.
- Really, this is partial application: some parameters to the expressions won't be known until later on.
- The funs add quite a lot of noise…

- All three cases show funs being created to supply parameters to simple expressions.
- Really, this is partial application: some parameters to the expressions won't be known until later on.
- The funs add quite a lot of noise…
- *How about changing the syntax? Lose some flexibility, but gain brievity…*

The meaning of _

- _ can already appear in patterns. That is unchanged.
- Cut allows _ to appear outside of patterns.
- Where a _ is found which isn't in a pattern, it becomes a parameter to the expression in which it *directly* appears.
- Multiple _s can appear in the same expression: multiple parameters

Cut in action

RabbitMQ

```
info_all(VHostPath, Items) ->
  map(VHostPath, fun (Q) -> info(Q, Items) end).
```

Cut in action

RabbitMQ

```
info_all(VHostPath, Items) ->
  map(VHostPath, fun (Q) -> info(Q, Items) end).

info_all(VHostPath, Items) -> map(VHostPath, info(_, Items)).
```

Cut in action

RabbitMQ

```
backing_queue_timeout(State = #q { backing_queue = BQ }) ->
  run_backing_queue(
    BQ, fun (M, BQS) -> M:timeout(BQS) end, State).
```

Cut in action

RabbitMQ

```
backing_queue_timeout(State = #q { backing_queue = BQ }) ->
  run_backing_queue(
    BQ, fun (M, BQS) -> M:timeout(BQS) end, State).

backing_queue_timeout(State = #q{backing_queue = BQ}) ->
  run_backing_queue(BQ, _:timeout(_), State).
```

Cut in action

RabbitMQ

```
reset_msg_expiry_fun(TTL) ->
  fun (MsgProps) ->
    MsgProps #message_properties {
      expiry = calculate_msg_expiry(TTL) }
  end.
```

```
reset_msg_expiry_fun(TTL) ->
  fun (MsgProps) ->
    MsgProps #message_properties {
      expiry = calculate_msg_expiry(TTL) }
  end.

reset_msg_expiry_fun(TTL) ->
  _ #message_properties { expiry = calculate_msg_expiry(TTL) }.
```

Cut controversy!

- Should there be a phantom cut marker to make it clear a cut is in use?

Cut controversy!

- Should there be a phantom cut marker to make it clear a cut is in use?

```
Menu = #menu{ breakfast = Toast, dinner = _ }
```

versus

```
Menu = cut(#menu{ breakfast = Toast, dinner = _ })
```

Cut controversy!

- Should there be a phantom cut marker to make it clear a cut is in use?

```
Menu = #menu{ breakfast = Toast, dinner = _ }
```

versus

```
Menu = cut(#menu{ breakfast = Toast, dinner = _ })
```

- Cut doesn't excuse poorly named variables! How about `MenuCtr` or `MenuFun` or `PendingDinner`?

Cut controversy!

RabbitMQ

- Why limit it to only shallow expressions?

```
NotAFun = {a, b, {c, _, e}}
```

Cut controversy!

RabbitMQ

- Why limit it to only shallow expressions?

```
NotAFun = {a, b, {c, _, e}}
```

- Cut is not a general purpose replacement of funs!

Cut controversy!

RabbitMQ

- Why limit it to only shallow expressions?
  `NotAFun = {a, b, {c, _, e}}`
- Cut is not a general purpose replacement of funs!
- However, time will tell!

Because a simple fun is being constructed by the cut, the arguments are evaluated prior to the cut function.

```
f1(_, _) -> io:format("in f1~n").

test() ->
  F = f1(io:format("test line 1~n"), _),
  F(io:format("test line 2~n")).
```

will print out

```
test line 2
test line 1
in f1
```

More cut examples

RabbitMQ

- Tuples

```
F = {_, 3},
{a, 3} = F(a).
```

More cut examples

RabbitMQ

- Lists

```
dbl_cons(List) -> [_, _ | List].

test() ->
  F = dbl_cons([33]),
  [7, 8, 33] = F(7, 8).
```

- Lists

```
dbl_cons(List) -> [_, _ | List].

test() ->
  F = dbl_cons([33]),
  [7, 8, 33] = F(7, 8).
```

- Lists in tail position are not sub-expressions!

  `A = [a, b | [c, d | [e]]]`

  is exactly the same (right from the Erlang parser onwards) as:

  `A = [a, b, c, d, e]`

More cut examples

RabbitMQ

- Records

```
-record(vector, { x, y, z }).

test() ->
  GetZ = _#vector.z,
  7 = GetZ(#vector { z = 7 }),
  SetX = _#vector{x = _},
  V = #vector{ x = 5, y = 4 } =
      SetX(#vector{ y = 4 }, 5).
```

- Case

```
F = case _ of
      N when is_integer(N) -> N + N;
      N                    -> N
    end,
10 = F(5),
ok = F(ok).
```

Cut conclusions

- Passing around funs is common: callbacks etc.
- Construction of those funs is frequently partial application of parameters to some simple expression.
- Cut helps make those cases less verbose.
- Cut also eases some pain of record syntax.

IMPORT AND IMPORT_AS ATTRIBUTES

- The `-import(my_module, [f/3, g/2, h/4])` attribute allows you to import `my_module:f/3`, `my_module:g/2` and `my_module:h/4` into the current module.
- You can then treat them as normal functions, local to the module.

IMPORT AND IMPORT_AS ATTRIBUTES

- The `-import(my_module, [f/3, g/2, h/4])` attribute allows you to import `my_module:f/3`, `my_module:g/2` and `my_module:h/4` into the current module.
- You can then treat them as normal functions, local to the module.
- But you can't import them with aliasing. Thus, this goes wrong:

```
-import(my_mod,       [size/1]).
-import(my_other_mod, [size/1]).
```

IMPORT AND IMPORT_AS ATTRIBUTES

- Solved!

```
-import_as({my_mod,       [{size/1, m_size}]})
-import_as({my_other_mod, [{size/1, o_size}]})
```

IMPORT AND IMPORT_AS ATTRIBUTES

- Solved!

```
-import_as({my_mod,       [{size/1, m_size}]})
-import_as({my_other_mod, [{size/1, o_size}]})
```

- Literally, we inject:

```
m_size(A) -> my_mod:size(A).
o_size(A) -> my_other_mod:size(A).
```

Thus you can use fun abstractions (or cuts!) on them, you can export them, etc: they are real local functions.

Monads: introduction

- Monads are widely used in Haskell, where they are essential in order to control sequencing of operations which may have side effects.
- They are not essential in Erlang.
- Monads provide very powerful control-flow and abstraction mechanisms which are of benefit to all languages.
- In Erlang, essentially, it's a programmatic comma!

Goal: control whether each statement is evaluated.

```
my_function() ->
    A = foo(),
    B = bar(A, dog),
    ok.
```

Goal: control whether each statement is evaluated.

```
my_function() ->
    A = foo(),
    comma(),
    B = bar(A, dog),
    comma(),
    ok.
```

Goal: control whether each statement is evaluated.

```
my_function() ->
    comma(foo(),
          fun (A) -> comma(bar(A, dog),
                           fun (B) -> ok end)).
```

Goal: control whether each statement is evaluated.

```
my_function() ->
    comma(foo(),
          fun (A) -> comma(bar(A, dog),
                           fun (B) -> ok end)).
```

As defined, the `comma/2` function is the monadic function `»=/2`. A monad needs only three functions: `»=/2`, `return/1` and `fail/1`.

1. Make syntax less awful

```
do([Monad ||
    A <- foo(),
    B <- bar(A, dog),
    ok]).
```

Readable, straightforward. Parse-transformer rewrites into:

```
Monad:'>>='(foo(),
            fun (A) -> Monad:'>>='(bar(A, dog),
                                   fun (B) -> ok end)).
```

2. Allow many implementations of Monads

- The `do`-block is parameterised by the type of monad we want to use.
- Within a `do`-block, calls to `return/1` and `fail/1` are rewritten to `Monad:return/1` and `Monad:fail/1`.

COMMAS, AS WE KNOW THEM

```
-module(identity_m).
-behaviour(monad).
-export(['>>='/2, return/1, fail/1]).

'>>='(X, Fun) -> Fun(X).
return(X)     -> X.
fail(X)       -> throw({error, X}).
```

COMMAS, AS WE KNOW THEM

```
-module(identity_m).
-behaviour(monad).
-export(['>>='/2, return/1, fail/1]).

'>>='(X, Fun) -> Fun(X).
return(X)     -> X.
fail(X)       -> throw({error, X}).

do([identity_m ||
    A <- foo(),
    B <- bar(A, dog),
    ok]).
```

COMMAS, AS WE KNOW THEM

```
-module(identity_m).
-behaviour(monad).
-export(['>>='/2, return/1, fail/1]).

'>>='(X, Fun) -> Fun(X).
return(X)     -> X.
fail(X)       -> throw({error, X}).

identity_m:'»='(
  foo(), fun (A) -> identity_m:'»='(
                      bar(A, dog), fun (B) -> ok end)).
```

COMMAS, AS WE KNOW THEM

```
-module(identity_m).
-behaviour(monad).
-export(['>>='/2, return/1, fail/1]).

'>>='(X, Fun) -> Fun(X).
return(X)     -> X.
fail(X)       -> throw({error, X}).

A = foo(),
B = bar(A, dog),
ok.
```

Maybe continue?

```
-module(maybe_m).
-behaviour(monad).
-export(['>>='/2, return/1, fail/1]).

'>>='({just, X}, Fun) -> Fun(X);
'>>='(nothing,  _Fun) -> nothing.

return(X) -> {just, X}.
fail(_X)  -> nothing.
```

Maybe continue?

```
-module(maybe_m).
-behaviour(monad).
-export(['>>='/2, return/1, fail/1]).

'>>='({just, X}, Fun) -> Fun(X);
'>>='(nothing,  _Fun) -> nothing.

return(X) -> {just, X}.
fail(_X)  -> nothing.
```

Do not continue if an expression returns `nothing`.

```
if_safe_div_zero(X, Y, Fun) ->
    do([maybe_m ||
        Result <-
            case Y == 0 of
                true  -> fail("Cannot divide by zero");
                false -> return(X / Y)
            end,
        return(Fun(Result))]).
```

```
if_safe_div_zero(X, Y, Fun) ->
    do([maybe_m ||
        Result <-
            case Y == 0 of
                true  -> fail("Cannot divide by zero");
                false -> return(X / Y)
            end,
        return(Fun(Result))]).

{just, 6} = if_safe_div_zero(10, 5, _+4)
```

```
if_safe_div_zero(X, Y, Fun) ->
    do([maybe_m ||
        Result <-
            case Y == 0 of
                true  -> fail("Cannot divide by zero");
                false -> return(X / Y)
            end,
        return(Fun(Result))]).

{just, 6} = if_safe_div_zero(10, 5, _+4)

nothing   = if_safe_div_zero(10, 0, _+4)
```

JUST LIKE MAYBE BUT KEEP THE ERROR

```
-module(error_m).
-behaviour(monad).
-export(['>>='/2, return/1, fail/1]).

'>>='({error, _Err} = Error, _Fun) -> Error;
'>>='({ok, Result},           Fun) -> Fun(Result);
'>>='(ok,                     Fun) -> Fun(ok).

return(ok) -> ok.
return(X)  -> {ok,    X}.
fail(X)    -> {error, X}.
```

Invisible error handling

```
Result = do([error_m ||
             Hdl <- file:open(Path, Modes),
             Data <- file:read(Hdl, BytesToRead),
             file:write(Hdl, DataToWrite),
             file:sync(Hdl),
             file:close(Hdl),
             file:rename(Path, Path2),
             file:delete(Path),
             return(Data)]).
```

INVISIBLE ERROR HANDLING

```
Result = do([error_m ||
             Hdl <- file:open(Path, Modes),
             Data <- file:read(Hdl, BytesToRead),
             file:write(Hdl, DataToWrite),
             file:sync(Hdl),
             file:close(Hdl),
             file:rename(Path, Path2),
             file:delete(Path),
             return(Data)]).
```

`Result` is always either `{ok, Data}` or `{error, Reason}`, regardless of where the failure happened. How many case statements would you need to achieve the same without monads?!

Going further: The State Transformer

RabbitMQ

- Monadic transformers embue monads with additional functionality.
- Imagine a monad within a monad, where the inner monad can reach out and interact with the outer monad.
- One such outer monad is the State transformer.
- This allows manipulation of state: `put` sets the current state, whilst `get` returns the current state. `modify` takes a function that takes the state and returns a new state.

Many numbered variables: Very Common, and Very Unpleasant

```
State1 = init(Dimensions),
State2 = plant_seeds(SeedCount, State1),
{DidFlood, State3} = pour_on_water(WaterVolume, State2),
State4 = apply_sunlight(Time, State3),
{DidFlood2, State5} = pour_on_water(WaterVolume, State4),
{Crop, State6} = harvest(State5),
```

AFTER APPLYING STATE TO IDENTITY

```
StateT = state_t:new(identity_m),
SM = StateT:modify(_),
SMR = StateT:modify_and_return(_),
StateT:exec(
  do([StateT ||
      StateT:put(init(Dimensions)),
      SM(plant_seeds(SeedCount, _)),
      DidFlood <- SMR(pour_on_water(WaterVolume, _)),
      SM(apply_sunlight(Time, _)),
      DidFlood2 <- SMR(pour_on_water(WaterVolume, _)),
      Crop <- SMR(harvest(_)),
      ]), undefined).
```

AFTER APPLYING STATE TO IDENTITY

```
StateT = state_t:new(identity_m),
SM = StateT:modify(_),
SMR = StateT:modify_and_return(_),
StateT:exec(
  do([StateT ||
      StateT:put(init(Dimensions)),
      SM(plant_seeds(SeedCount, _)),
      DidFlood <- SMR(pour_on_water(WaterVolume, _)),
      SM(apply_sunlight(Time, _)),
      DidFlood2 <- SMR(pour_on_water(WaterVolume, _)),
      Crop <- SMR(harvest(_)),
      ]), undefined).
```

Look! No numbered state variables!

Do conclusions

- Monads are very powerful and flexible.
- Takes some practise to get used to, and harder due to lack of useful type checker.
- Do-notation essential to making monads at all pleasant.
- Implementation very similar to Haskell, so mechanical translation of Haskell's libraries quite possible.

What's coming next?

- Type classes (value based dynamic dispatch)
- Ability to define infix functions
- Convenience mechanisms for records, e.g.

```
#state { foo, bar, baz }
≡ #state { foo = Foo, bar = Bar, baz = Baz }
```

- Whole module importing with aliasing

Where can I get this magical sauce?

`http://hg.rabbitmq.com/erlando` or
`http://github.com/rabbitmq/erlando`

Thank you

Questions?