Erlang *Dos* and *Don'ts*

Richard Carlsson

Klarna

Prefer to avoid macros

- **Don't:**

```
-define(msg(X), {message, self(), X}).
```

- **Do:**

```
msg(X) → {message, self(), X}.
```

- **Why:**

  • Macros shared between modules require recompiling of all modules if you make a change

  • You get better compiler warnings with functions

  • The compiler can handle inlining for you

Macros should be well formed

- **Don't:**

```
-define(p(X), io:write(X), ).
-define(do(X), fun () -> ?p(X) ).
-define(done(X), ?p(X) ok end ).
```

- **Do:**

```
% Something else - anything else
```

- **Why:**
  - Screws up readability of the code
  - Makes it impossible to parse in EDoc and other tools that try to analyze the source code
  - There is a circle in hell reserved for those who use macros this way

Do not export everything

- **Don't:**

```
-compile(export_all).
```

- **Do:**

```
-export([…]).
```

- **Why:**
  - Clear distinction between official API and internal support functions
  - Lets the compiler make better optimizations
  - Helps Dialyzer make better type analysis

List all cases explicitly

- **Don't:**

```
case is_foo(X) of
    true -> ... ;
    _ -> ...
end
```

- **Do:**

```
case is_foo(x) of
    true  -> ... ;
    false -> ...
end
```

- **Why:**
  - If there's a bug in is_foo(X) that makes it return a non-boolean value sometimes, you will not notice it, because the catch-all case will always be selected

Be careful with list_to_atom()

- **Don't:**

```
foo(InputString) ->
    … list_to_atom(InputString)
```

- **Do:**

```
foo("on") -> on;
foo("off") -> off;
    …
```

- **Why:**
  - The atom table in Erlang is limited in size, and is never garbage collected
  - A malicious client could send random strings until the table is full, crashing our node
  - list_to_existing_atom(String) can also be used

Don't create improper lists

- **Don't:**

```
L1 = [L | 42]   % 42 as "tail"
```

- **Do:**

```
L1 = L ++ [42]  % if you have to

L1 = [L, 42]    % if a deep list is ok
```

- **Why:**
  - Almost all functions that expect a list will crash if the list is not proper (i.e., does not end with [ ])
  - Even if you don't care right now, someone may want to use a normal list function on L1 later
  - It will make type analysis in Dialyzer less exact

Avoid length(List)

- **Don't:**

```
foo(L) when length(L) > 0 ->
    …
```

- **Do:**

```
foo([_|_]=L) ->
    …
```

- **Why:**
  - length(L) has to traverse the entire list L
  - If you do this within a loop over L, it will cause quadratic time complexity
  - Pattern matching instead takes constant time

Don't append to the right

- **Don't:**

```
foo([Ls | Lss], …, As) ->
    … foo(Lss, …, As ++ Ls);
foo([], As) ->
    As.
```

- **Do:**

```
foo([Ls|Lss], …, As) ->
    … foo(Lss, …, reverse(Ls) ++ As);
foo([], As) ->
    reverse(As).
```

- **Why:**
  - A ++ B creates a new list by copying A onto B, which takes time proportional to the length of A
  - If you do this within a loop over A, it will cause quadratic time complexity

Avoid --

- **Don't:**

```
Rest = BigList -- AnotherBigList
```

- **Do:**

```
S1 = ordsets:from_list(List1),
S2 = ordsets:from_list(List2),
Rest = ordsets:subtract(S1, S2)
```

- **Why:**
  - A -- B searches and rewrites A once for each element in B, so it runs in O(A*B) time
    - If B is guaranteed to be short, this is OK
  - [1,2,2] -- [2] → [1,2]
    - If it's necessary to preserve order and repeated elements in A, then -- can be OK

Updating tuples means copying

- **Don't:**

```
T1 = setelement(999, T0, Value)
```

- **Do:**

```
A0 = array:new(),
A1 = array:set(999, Value, A0)
```

- **Why:**
  - Updating a tuple of size N copies N machine words of data, just to replace a single word
  - The array module is better if N is large (>50), and it also handles sparse arrays well
  - Remember that records are tuples

Use the right size operator

- **Don't:**

```
foo(T, B) when size(T) > 2 ->
    Bytes = size(B)
```

- **Do:**

```
foo(T, B) when tuple_size(T) > 2 ->
    Bytes = byte_size(B),
    Bits = bit_size(B)
```

- **Why:**
  - size(X) is too overloaded (accepts both tuples and binaries), giving no clues to the reader, nor to the compiler or Dialyzer
  - Note that if e.g. B = <<255, 255, 17:3>>, then bit_size(B) is 19, and byte_size(B) is 3 – the smallest number of bytes needed

Avoid is_record()

- **Don't:**

```
foo(R) when is_record(R, bar) ->
   … X = R#bar.foo
```

- **Do:**

```
foo(#bar{foo=X}=R) ->
   …
```

- **Why:**
  - More obvious to the reader
  - Easier to find uses of #bar with grep etc.
  - Better hints to the compiler

Store text as binaries

- **Don't:**

```
% huge dict of strings in server state
D1 = dict:store(Key, String, D0)
```

- **Do:**

```
% encode as UTF-8 (accepts deep lists)
B = unicode:characters_to_binary(Txt)
```

- **Why:**
  - A string (list of characters) uses 8 bytes per character in RAM (16 bytes on 64-bit machines)
  - UTF-8 binary strings are much more compact
  - Use strings as temporary buffers and use binaries for more long-term storage
  - BUT: not worth the effort if it's not a lot of data

Use I/O-lists for output

- **Don't:**

```
Out = L1 ++ L2,
io:format("Result: ~s\n", [Out])
```

- **Do:**

```
Out = [L1, L2],
io:format("Result: ~s\n", [Out])
```

- **Why:**
  - All functions that send data to output streams accept I/O-lists as data
  - It's a complete waste of time to create a flat list if you don't really need it to be flat
  - No need to turn binaries to lists first – you can mix binaries and character codes in I/O-lists

Parameter type overload

- **Don't:**

```
foo(X) when is_integer(X) ->
    -X;
foo(X) when is_float(X) ->
    1 / X.
```

- **Do:**

```
negate(X) -> -X.

inverse(X) -> 1 / X.
```

- **Why:**
  - The reader doesn't have to know the type of X to know what the function does
  - Allow compiler to make better optimizations and Dialyzer to make better type analysis

Confusing types

- **Don't:**

```
% integer for age or string for name
-spec f() -> integer() | string().
```

- **Do:**

```
-spec f() -> {age, integer()}
           | {name, string()}.
```

- **Why:**
  - Tagging makes representation self-explaining
  - Much easier to debug
  - Better type analysis in Dialyzer

Avoid leaking data representation

- **Don't:**

```
-spec f() -> {user, atom(), string()}.
```

- **Do:**

```
-spec f() -> {user, [Info]}
  when Info :: {name, string()}
             | {id, atom()}
```

- **Why:**
  - Makes it hard to change representation later
  - Is all the code that knows about the representation under your control?
  - How many lines of code need changing, and in how many different modules?
  - What about data stored on disk, or in an ETS table, or in the state of a server process?

Spawn and link

- **Don't:**

```
spawn(worker, start, [Data])
```

- **Do:**

```
spawn_link(worker, start, [Data])
```

- **Why:**
  - Ensures all linked processes get cleaned up, if one of them crashes
  - Orphaned processes can be hard to track down
  - Only spawn without linking if you *know* that is exactly what you want to do
  - Try to ensure that unlinked processes are registered somehow, so you can find them later

Make code upgrade possible

- **Don't:**

```
loop(…) ->
    receive
        … -> … loop(…)
    end.
```

- **Do:**

```
loop(…)->
    receive
        … -> … ?MODULE:loop(…)
    end.
```

- **Why:**

  • If you only loop using local calls, the process will not switch to the new code when you reload the module. Reloading twice will kill the process.

Anonymous funs should be short-lived

- **Don't:**

```
F = fun (X) -> X + N end,
ets:insert(Table, {Key, F})
```

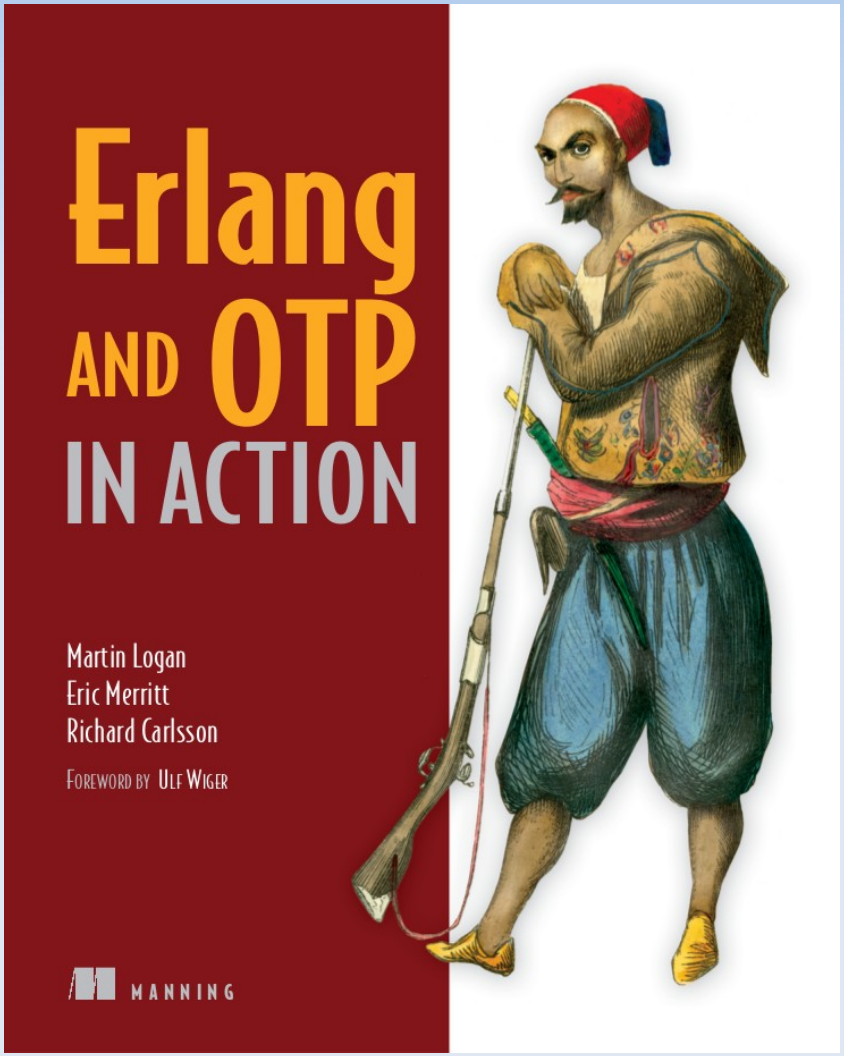
- **Do:**

```
F = fun mymod:myfun/1,
ets:insert(Table, {Key, F, N})
```

- **Why:**
  - If the module that the anonymous fun belongs to gets reloaded more than once, the existing fun-values will become invalid and will cause the processes that hold them to be killed
  - Named funs do not have this effect

OTP

Read it!


Erlang
AND OTP
IN ACTION
Martin Logan
Eric Merritt
Richard Carlsson
Foreword by Ulf Wiger
MANNING

Use the application environment

- **Don't:**

```
init([]) ->
    {ok, #state{port = 1234}}.
```

- **Do:**

```
init([]) ->
    Port = application:get_env(port),
    {ok, #state{port = Port}}.
```

- **Why:**
  - Makes it easy to reconfigure your application via the system config file.
  - Inject dependencies between applications via configuration, instead of hard-coding them
  - Preferably, use get_env at startup time only, and use messages for dynamic reconfiguration

Use gen_server

- **Don't:**

```
% home made server loop
start(S) -> spawn(?MODULE, loop, [S]).
loop(S0) ->
    receive
        …
        loop(S1)
    end.
```

- **Do:**

```
start(…) -> gen_server:start_link(…).
```

- **Why:**
  - Predictability, easier to understand and debug.
  - Don't repeat the same old mistakes as everybody else (at least not on company time)

Use supervisors dynamically

- **Don't:**

```
% non-supervisor spawning a worker
spawn_link(worker, run, [Data])
```

- **Do:**

```
% use a 'simple_one_for_one' supervisor
% as a process factory with supervision

supervisor:start_child(worker_sup, [Data])
```

- **Why:**
  - Easier to control, debug, and visualize in tools
  - Better crash reports
  - Automatic restarts

Use proc_lib to spawn

- **Don't:**

```
spawn_link(worker, run, [Data])
```

- **Do:**

```
% if you really want to use spawn
% (mainly for simple background jobs)

proc_lib:spawn_link(worker, run, [Data])
```

- **Why:**
  - Sets up OTP environment in the new process (certain process dictionary settings used by OTP)
  - Better crash reports via SASL
  - Easier for OTP tools to visualize etc.

Keep logic out of supervisors

- **Don't:**

```
init([]) ->
    application:start(crypto),
    Children = case … of
    …
```

- **Do:**

```
init([]) ->
    Children = […],
    {ok, {{one_for_one,3,10}, Children}}.
```

- **Why:**
  - If something stops working, your whole application might fail to start, or cannot restart
  - This kills the node
  - Preferably, don't even call other modules

The init() callback must finish fast

- **Don't:**

```
init([LSocket]) ->
    {ok, S} = gen_tcp:accept(LSocket),
    {ok, #state{socket=S}}.
```

- **Do:**

```
% this uses the server timeout trick
init([A, B]) ->
    {ok, #state{a=A, b=B}, 0}.

handle_info(timeout, State) -> ...
```

- **Why:**
  - The process trying to start your gen_server can time out and crash if init() takes too long time

Make handle_...() return quickly

- **Don't:**

```
handle_call({find, X}, _From, St) ->
    {reply, {ok, search_db(X)}, St}.
```

- **Do:**

```
handle_call({find, X}, From, St) ->
    proc_lib:spawn_link(fun ()->
          Reply = {ok, search_db(X)},
          gen_server:reply(From, Reply)}
        end),
    {noreply, St}.
```

- **Why:**
  - Blocks the gen_server process from handling other requests until the computation is done

Maybe make it asynchronous

- **Don't:**

```
handle_call({prompt, P}, _From, St) ->
    {reply, {ok, user_input(P)}, St}
```

- **Do:**

```
handle_cast({prompt, P, From}, St) ->
    proc_lib:spawn_link(fun ()->
          From ! {prompt, user_input(P)},
        end),
    {noreply, St}.
```

- **Why:**
  - The caller is blocked, and could time out and crash if the call takes a long time.
  - Cast + Reply is more complicated for the caller, but lets it do other things while waiting.

Here endeth the lesson