Who Doesn't Love Emoji 💖

By Ken Friis Larsen (`kflarsen@diku.dk`)

Last revision: September 28, 2021

Part 1: Warm up

The Haskell package async provide a (monadic) library for running IO operations asynchronously and waiting for their results. Implement an Erlang module, `async`, to provide similar functionality. That is, you should implement the following minimal API, where `Aid` stands for an action ID, which is an opaque datatype (that is, you decide what it should be):

- `new(Fun, Arg)` that starts a concurrent computation that computes `Fun(Arg)`. It returns an action ID.
- `poll(Aid)` that check whether an asynchronous action has completed yet. If it has not completed yet, then the result is `nothing`; otherwise the result is either `{exception, Ex}` if the asynchronous action raised the exception `Ex`, or `{ok, Res}` if it returned the value `Res`.
- `wait(Aid)` that waits for an asynchronous action to complete, and return the value of the computation. If the asynchronous action threw an exception, then the exception is re-thrown by `wait`.

Part 2: Emoji Server

Objective

The learning objectives of this assignment are:

- Gain hands-on programming experience with Erlang, how to structure modules and splitting code into functions.
- Learn how to implement a simple server.
- Dipping the toes into making robust code.

The assignment is about making an *Emoji server* that allow looking up shortcodes for emoji.

Terminology

Emoji (aka emoticons) are ideograms and smileys used in electronic messages and web pages.

An *emoji shortcode* is an ASCII string starting and ending with `:` (colon) that denotes an emoji. For instance, `:smiley:` denotes 😃, and `:facepalm:` denotes 🤦.

We represent emoji as binaries with the UTF-8 encoding of the emoji. For instance the binary `<<240,159,152,131>>` represents 😃 and `<<240,159,164,166>>` represents 🤦.

An analytics function is a function that takes a shortcode and some state and returns an updated state. Analytics functions may raise exception and may fail to terminate. Your code should be robust against this. Robust here means that an emoji server should, ideally, continue to run and answer requests, even if an analytics function throws exceptions or takes a long (potentially infinite) time to terminate. (Note that it is unspecified how to deal with **errors**, thus if you do something special you should document that.)

Types:

```
-type shortcode() :: string().
-type emoji() :: binary().
-type analytic_fun(State) :: fun((shortcode(), State) ->
        State).
```

In this assignment we shall not be concerned with enforcing a specific form of shortcodes, such as starting and ending with `:`, this a job for a parser using our module. Any string will do. Likewise, it's out of scope to enforce that the binaries are valid UTF-8 encoded emoticons.

The Emoji module

Implement an Erlang module `emoji` with the following API, where `E` always stands for an Emoji server:

- `start(Initial)` 😄 for starting an Emoji server, with the initial shortcodes definition given by `Initial`. The type of `Initial` should be a list of pairs, where the first component is a string and the second is a binary. All shortcodes in `Initial` should be unique, otherwise it is an error.

  Returns `{ok, E}` on success, or `{error, Reason}` if some error occurred.

- `new_shortcode(E, Short, Emo)` 😃 for registering that the shortcode `Short` denotes the emoji `Emo`. It is an error to try to register the same shortcode more than once.

  Returns `ok` on success or `{error, Reason}` if some error occurred.

- `alias(E, Short1, Short2)` 🙌 for registering `Short2` as an alias for `Short1` . It is an error if `Short1` is not registered, or if `Short2` is already registered.

  After this function succeeds `Short2` should be a valid shortcode in every sense.

  Returns `ok` on success or `{error, Reason}` if some error occurred.

- `delete(E, Short)` 💥 removes any denotations for `Short`. This is a *non-blocking* function that should always succeed (thus the return value is unspecified). Should delete all aliases for `Short` as well.

- `lookup(E, Short)` 👀 for looking up the emoji for `Short`.

  Returns `{ok, Emo}` on success, or `no_emoji` if no shortcode (including aliases) are found.

- `analytics(E, Short, Fun, Label, Init)` 📈 for registering an analytics function for `Short` (and aliases). Where `Fun` is an analytics function, `Init` is the initial state for `Fun` (thus `Init` could be anything), and `Label` is a string.

  A shortcode can have multiple analytics functions registered, but each must have a unique label. Analytics functions are evaluated when a shortcode (including aliases) is looked up.

  Returns `ok` on success or `{error, Reason}` if some error occurred.

- `get_analytics(E, Short)` 📉 for getting the result of all analytics functions associated with a specific shortcode (including aliases).

  Returns `{ok, Stat}` on success, where `Stat` is a list `[{string, any()}]` with the label and the (updated) state for each analytics function registered for `Short`; otherwise returns `{error, Reason}` if some error occurred.

- `remove_analytics(E, Short, Label)` 🔪 for removing the analytics function registered under `Label` for `Short` (including aliases). This is a non-blocking function.

- `stop(E)` :godmode: for stopping an emoji server, and all associated processes (if relevant). Any use of `E` after this function has been called is an error that is outside the scope of the library.

Returns `ok` once all processes associated with `E` has terminated; or `{error, Reason}` if some error occurred.

Example use of the `emoji` module

The following example module shows how to use the `emoji` module:

```
-module(love_emoji).
-export([try_it/0]).

hit(_, N) -> N+1.
accessed(SC, TS) ->
  Now = calendar:local_time(),
  [{SC,Now} | TS].

setup() ->
    {ok, E} = emoji:start([]),
    ok = emoji:new_shortcode(E, "smiley",
        <<240,159,152,131>>),
    ok = emoji:new_shortcode(E, "poop",
        <<"\xF0\x9F\x92\xA9">>),
    ok = emoji:alias(E, "poop", "hankey"),
    ok = emoji:analytics(E, "smiley", fun(_, N) -> N+1 end,
        "Counter", 0),
    ok = emoji:analytics(E, "hankey", fun hit/2, "Counter",
        0),
    ok = emoji:analytics(E, "poop", fun accessed/2,
        "Accessed", []),
    E.

print_analytics(Stats) ->
    lists:foreach(fun({Lab, Res}) -> io:fwrite("  ~s: ~p~n",
        [Lab, Res]) end,
                  Stats).

try_it() ->
    E = setup(),
    {ok, Res} = emoji:lookup(E, "hankey"),
    io:fwrite("I looked for :hankey: and got a pile of
        ~ts~n", [Res]),
    {ok, Stats} = emoji:get_analytics(E, "poop"),
    io:fwrite("Poppy statistics:~n"),
    print_analytics(Stats),
    io:fwrite("(Hopefully you got a 1 under 'Counter')~n").
```

Testing `emoji`

You tests should be in a module called `test_emoji` in the `tests` directory, this module should export at least one function, `test_all/0`.

To test the quality of your test-suite, we might run your tests against our special `emoji` module(s), and OnlineTA might not do any testing of your code unless it is minimally satisfied with your tests.

Note that since we'll run your tests against *our* `emoji` module, the tests that are performed from `test_all/0` shouldn't rely on things specific to *your* solution. Thus, if you have tests that are specific to your implementation, then you might want to export a `test_everything/0` function as well (that could call `test_all/0`).

The `code/part2/tests/someemoji.erl` module contain two functions that return a list of pairs, where the first component is a string and the second is a binary, that you can use for your testing.

How to get started

The two difficult parts of this assignment are how to deal with aliases and how to deal with analytics function. Thus, the recommended course of action, is to first get a minimal version working that doesn't include aliases nor analytics function. After that, to deal either aliases or analytics functions, but not both at the same time.

Somewhat counterintuitive, analytics functions can be easier to start with before trying to deal with aliases.

Trickiness of aliases

Aliases can be hard to wrap your head around. Especially `delete/2` can be hard to get right. Thus, you might want to start by writing some tests to help you understand the issues.

For instance, when you delete a shortcode that has one or more aliases then all aliases should be deleted, you should never need to look at the binary/binaries. Observe, that there is a difference between

```
ok = emoji:new_shortcode(E, "poop", <<"\xF0\x9F\x92\xA9">>),
ok = emoji:alias(E, "poop", "hankey")
```

and

```
ok = emoji:new_shortcode(E, "+1", <<240,159,145,141>>),
ok = emoji:new_shortcode(E, "thumbsup", <<240,159,145,141>>)
```

In the first case you have an alias in the second you don't. In the first case, if you delete either `poop` or `hankey` both are gone. In the second case, if you delete `thumbsup` them you still have `+1` as a shortcode.

Hand-in Instructions

You should hand in two things:

1. A short report, `report.pdf` (normally 1–3 pages, including figures), for the main (not warm-up) part only, explaining the main design of your code, and **an assessment** of your implementation, including what this assessment is based on. See the generic instructions regarding the report in previous assignments.

   In this assignment, it is of particular interest that you discuss how you deal with unreliable analytics functions. You should shortly discuss what issues they cause and how you deal with them. You should also describe what state your server (and other relevant processes) maintain. Likewise, consider making a diagram of your processes. How do the processes interact, and how many (kinds) are there in the system?

2. A ZIP archive `code.zip`, mirroring the structure of the handout, containing your timesheet, source code and tests.

   Your own tests should be separated in one or more test files that must be in the `tests` directory, so that OnlineTA can treat them right.

Make sure that you adhere to the types of the API, and test that you do.

To keep your TA happy, follow these simple rules:

1. The Erlang compiler, with the parameter `-Wall`, should not yield any errors or warnings for your code.
2. You should comment your code properly, especially if you doubt its correctness, or if you think there is a more elegant way to write a certain piece of code. A good comment should explain your ideas and provide insight.
3. Adhere to the restrictions set in the assignment text, and make sure that you export all of the requested API (even if some functions only have a dummy implementation).
4. Describe clearly what parts of the assignment you have solved.

End of assignment text.